The Impact of Meta-Tracing on VM Design and Implementation

Carl Friedrich Bolz[a], Laurence Tratt[b]

[a]*Heinrich-Heine-Universität Düsseldorf, 40204 Düsseldorf, Germany.*
[b]*King's College London, Strand, London, WC2R 2LS, United Kingdom.*

Abstract

Most modern languages are implemented using Virtual Machines (VMs). While the best VMs use Just-In-Time (JIT) compilers to achieve good performance, JITs are costly to implement, and few VMs therefore come with one. The RPython language allows tracing JIT VMs to be automatically created from an interpreter, changing the economics of VM implementation. In this paper, we explain, through two concrete VMs, how meta-tracing RPython VMs can be designed and optimised, and the performance levels one might reasonably expect from them.

Keywords:
Virtual machines, meta-tracing, programming languages.

1. Introduction

Every programming language that makes it beyond a paper design needs a corresponding implementation. Traditionally, most languages were compiled to machine code (via assembler or, less commonly, C). For languages with high static content (i.e. those with minimal runtimes) it can lead to highly efficient implementations, though it requires significant manpower to do so. However, languages with high dynamic content (i.e. those with complex runtimes, which includes most dynamically typed languages [1]) lack the static information needed to produce efficient code. In short, the traditional

Email addresses: `cfbolz@gmx.de` (Carl Friedrich Bolz), `laurie@tratt.net` (Laurence Tratt)

approach is generally either too costly, or leads to slow execution. For many modern languages, it is both.

It is now common for such languages to split apart the concept of compiler and 'execution engine', with the latter implemented as a Virtual Machine (VM). A simple compiler is all that is needed in such an approach, as the VM specifies most of the language's behaviour. Implemented as a naive interpreter, a VM will tend to have poor performance. Instead, a VM can use information about the way the program executes to optimise it in ways that are statically impossible. Self first showed how a carefully crafted VM could substantially improve performance for a highly dynamic language [2] and its influence has led to most new languages being implemented using VMs. For example, programs running on Java's HotSpot VM [3] (one of several Java VMs, henceforth referred to generically as 'JVMs'), which is derived in part from the Self VM, can often match C's performance.

However, VMs reflect the languages, or group of languages, they were designed for. If a language fits within an existing VMs mould, that VM will probably be an excellent target; if not, the *semantic mismatch* between the two leads to poor performance for user programs. For example, despite the often spectacular performance of HotSpot, Jython (Python for JVMs) almost never exceeds CPython (Python running with a custom VM written in C) in performance, and is generally slower because features that make use of Python's run-time customisability have no efficient implementation on a JVM. Similar problems have been observed when trying to implement Scheme on the JVM [4]. An attempt to improve in this regard is the introduction of the `invokedynamic` bytecode [5], but it remains to be seen how much it helps dynamic language implementation in practice and how easy it is to apply.

Thus, languages which do not fit an existing VM mould must look to a custom VM to achieve good performance levels. However, this is not easily achieved: implementing a performant VM is justifiably seen as requiring highly specialised skills and significant manpower. In particular, the highest performing VMs rely on Just-In-Time compilers (henceforth referred to as JITs), which, at run-time, takes part of a program and converts it to optimised machine code. Few VM teams have the resources to create a JIT, particularly for complex languages. Unfortunately, therefore, most VMs' performance is substantially below that of HotSpot or .NETs CLR.

In this paper, we consider the RPython language, which automatically creates JITing VMs from implementations of traditional interpreters through *meta-tracing*. This allows high-performance custom VMs to be created with reasonable resources. Many of the low-level details of RPython have been described elsewhere (see e.g. [6]). In this paper, we look at how a high-level VM is implemented in RPython, and the trade-offs involved, using two different VMs: PyPy [7] (a new VM for Python) and Converge [8]. The two VMs have very different aims: PyPy is a drop-in replacement for the standard VM; Converge is a VM for a research language. Consequently, the two have had different levels of effort put into them (PyPy about 60 man months; Converge about 3). Our aim in this paper is to explain, using Converge and PyPy as concrete examples, how RPython VMs can be designed and optimised, and what performance levels one might reasonably expect from them. This paper is the first to consider: specific RPython VM designs; the general lessons one can learn from them; and the effects of different man-power levels on such VMs.

We start looking at the languages that each VM implements (Section 2), before introducing the RPython language itself (Section 3). RPython produces meta-tracing JIT, so we give an introduction to this area for the many readers who are likely to be unfamiliar with it (Section 4). We then detail the high-level design of the VMs themselves (Section 5), explaining them as if they were traditional interpreters. From that follows the technical heart of the paper, explaining the optimisation techniques the VMs use to generate performant RPython JITs (Section 6), as well as the general lessons embodied in the specific techniques. To make concrete the lessons learned from the optimisations, we present the results of a performance comparison of the VMs (along with various other open-source VMs) to show where RPython VMs fit into the performance landscape (Section 7). Finally, we look at the issues and limitations of tracing and meta-tracing JITs (Section 8).

2. Python and Converge

Python and Converge are two seemingly similar languages. Both are dynamically typed, object orientated, have an indentation-based syntax, and a rich collection of built-in datatypes. Indeed, much of Converge was explicitly influenced by Python. In this paper, we assume a passing familiarity with Python (pointing out differences with Converge on an as-needs basis).

Two technical features in particular distinguish Converge from Python. First, Converge allows compile-time meta-programming whereby code can be executed and generated at compile-time [8] (for the purposes of this paper, this can be thought of as approximately equivalent to Lisp macros). Second, Converge's expression evaluation system is based on Icon's and can perform limited backtracking. This second feature is interesting from the implementation point of view because it imposes a noticeable performance penalty even on programs which make little or no use of it [9].

Perhaps more importantly, at a 'social' level, the two VMs have different motivations. Python is a real-world language, used by hundreds of thousands of developers worldwide for a huge number of tasks, and for which many external libraries are available. PyPy's goal is first to be fully compatible (warts and all) with CPython, and then to be faster. Converge on the other hand is a research language, intended to explore research on Domain Specific Languages (DSLs) and compile-time meta-programming. The needs of each language's VMs reflect this: PyPy strives to be as fast as possible; Converge strives to be 'fast enough'.

3. RPython

The basic facts about RPython are that it is strict subset of Python whose programs are translated to C. Every RPython program is a valid Python program and can also be run using a normal Python interpreter. However, RPython is suitably restricted to allow meaningful static analysis. Most obviously, static types (with a type system roughly comparable to Java's) are inferred and enforced. In addition, extra analysis is performed e.g. to assure that list indices are not negative. Users can influence the analysis with `assert` statements, but otherwise it is fully automatic. Unlike seemingly similar languages (e.g. Slang [10] or PreScheme [11]), RPython is more than just a thin layer over C: it is, for example, fully garbage collected and has several high-level datatypes.

In addition to outputting optimised C code, RPython automatically creates a second representation of the user's program. Assuming RPython has been used to write an interpreter for language L, one gets not only an optimised version of that interpreter, but also an optimising tracing Just-In-Time (JIT) compiler for under 10 lines of code [12]. In other words, when a program written in L executes on an appropriately written RPython VM, hot loops (i.e. those which are executed frequently) are automatically turned into machine code and executed directly.

RPython is able to do this because of the particular nature of interpreters. An interpreter, whether it be operating on bytecode or ASTs, is a large loop: 'load the next instruction, perform the associated actions, go back to the

User program	Trace when x is set to 6	Optimised trace
`if x < 0:`	`guard_type(x, int)`	`guard_type(x, int)`
`  x = x + 1`	`guard_not_less_than(x, 0)`	`guard_not_less_than(x, 0)`
`else:`	`guard_type(x, int)`	`x = int_add(x, 5)`
`  x = x + 2`	`x = int_add(x, 2)`	
`x = x + 3`	`guard_type(x, int)`	
	`x = int_add(x, 3)`	

Figure 1: An example of a user program and resulting traces.

beginning of the loop.' In order to switch from interpretation to JITing, RPython needs to know when a hot loop has been encountered, in order to generate machine code for that loop and to use it for subsequent executions. In essence, one need only add two annotations in the form of function calls to an RPython program to add a JIT. The first function call `can_enter_jit` is used to inform RPython that a loop in the user program has been encountered, and that it might want to start generating machine code if that loop has been encountered often. The second function call `jit_merge_point` is used to indicate to RPython when it can use an existing machine code version of a loop rather than using standard interpretation.

4. Tracing JITs

Traditional JITs are *method JITs*: when a particular method is identified as being 'hot', it is translated into machine code (leaving most of its control structures intact).

In contrast, the JITs that RPython creates are *tracing JITs*. Tracing JITs came to prominence in the Dynamo project [13] as well as Franz and Gal's work [14]. The basic idea behind tracing JITs is to identify hot loops, record the bytecodes taken during a specific execution of it ('the trace'), optimise the trace, and then convert that into machine code. Traces intentionally linearise control structures. Wherever a specific branch was taken, a *guard* (roughly speaking, a 'check') is inserted into the trace; if, during execution of the machine code version, a guard fails, execution returns to the interpreter. Traces are hoped to be records of commonly taken paths through a program; when that assumption holds true, the result is extremely fast execution.

Figure 1 shows a high-level example of a program and its trace. The left-hand column shows a user program written in a Python-like language.

When it is detected to be in a hot loop, the next time the code is executed, a trace is recorded. The middle column shows the trace recorded when `x` is set to 6. Note that the specific value of `x` is not recorded in the trace; indeed the trace would have been identical for any value of `x` greater than or equal to `0` (since the 'else' branch of the `if` would be taken for all such values); but the trace would be different if `x` was less than `0` (as the 'then' branch would be taken) or if `x` was not an integer. Once the trace has been recorded, the trace optimiser then attempts to reduce it in size, so that the resulting machine code executes as fast as possible. In this case, two type checks which are trivially true can be removed, and the two constant integer additions can be constant-folded. The resulting optimised trace is shown in the right-hand column.

Whereas tracing JITs are normally separate components from interpreters, RPython is a meta-tracing system [12]. The RPython translator in fact outputs two interpreters: the *language interpreter* is the (conceptually) simple translation of the RPython interpreter into C; the *tracing interpreter* is a second representation of the interpreter which can be run to create traces. When a hot loop in a user program is detected, a marker is left such that the next time the loop is about to run, the VM will use the tracing interpreter instead of the language interpreter. When the loop is next encountered, a complete execution of the loop is performed and each low-level action taken by the tracing interpreter is recorded. After the loop has finished, the trace is then analysed, optimised, and converted into machine code. All subsequent executions of the loop will then call the machine code version. RPython automatically inserts guards into the machine code to detect divergence from the machine code version's capabilities. If a guard fails at any point, execution falls back to the tracing interpreter for the rest of that bytecode, and then back to the language interpreter[1].

The fundamental difference between traditional tracing JITs and an RPython JIT is that traditional tracing JITs must be manually written. By tracing the actions the interpreter itself takes, a meta-tracing JIT can automatically create a JIT from the interpreter. As we shall see, the way an RPython interpreter is written effects the performance of the resulting JIT. To obtain the highest possible performance, the interpreter often needs to be subtly

[1]The reason that the tracing interpreter is only run sparingly is that it is extremely slow in comparison to the language interpreter.

rewritten in specific places to aid the resulting JIT. When this is done intelligently, the raw traces created by an RPython JIT will often be reduced by 90% by RPython's trace optimiser [15].

5. PyPy and Converge VM overview

In this section we give an overview of the structure of both VMs.

5.1. Bytecode structure

Both PyPy and Converge use a bytecode based interpreter together with a compiler that translates programs into the respective bytecode set. The bytecode sets are similar in intent, being stack-based and deferring type specialization until run-time. Python's bytecode set contains more instructions to optimise specific common operations (e.g. list accesses) than Converge, while the latter has several additional instructions related to backtracking.

5.2. Compilation

Both VMs store bytecode files (internally and in memory) in custom binary formats, for efficiency of size (being approximately 5-10x smaller than an equivalent XML format, for example) and execution by the eventual interpreter. Although both PyPy and Converge use traditional compilation, the implementations differ. PyPy's compiler is written in RPython and is integrated into the VM; most users will never be aware that separate compilation is performed on their behalf. Converge's compiler is written in Converge, and is a separate program; while it can be invoked manually, if passed a source file, the VM transparently calls the compiler. Both systems attempt to transparently cache bytecode output on disk to lower compiler costs, automatically recompiling any source files which are detected to be newer than their cached equivalents (this means that the first run of a Python or Converge program can be significantly slower than subsequent runs).

Because the Converge VM is used to compile new versions of the Converge compiler, the latter has to obey an important restriction: neither the compiler nor any libraries it uses can perform compile-time meta-programming. If the compiler were to do so, it would be impossible to migrate Converge's bytecode format, as the running compiler would then emit bytecode in the new VM format and attempt to execute it, all while still running on the old VM. In practise, this restriction is not particularly onerous, although it requires a freshly unpacked Converge system to be compiled in a specific order: first a minimal version of the standard library (enough for the compiler); then the compiler itself; then the full library (which, at this point, may include compile-time meta-programming).

5.3. Interpreter structure

Both PyPy and Converge split their interpreters into three major parts: the bytecode interpreter; the built-in datatypes; and the built-in libraries. The bytecode interpreters are responsible for dispatching and implementing the bytecode set and are constructed in a direct, simple fashion. The built-in datatypes realize basic concepts such as objects, classes, lists, and dictionaries. As well as being used extensively throughout the VM, several of these datatypes require careful bootstrapping during VM initialization. Built-in libraries are provided either for performance reasons or to allow integration with low-level C libraries.

In Converge, the split between the bytecode interpreter and built-in datatypes is relatively informal, as befits a simple VM. In PyPy, in contrast, the split is very clearly defined to ensure that, despite the large size of the Python language specification, the components are manageable. The bytecode interpreter treats all Python objects that it handles as black boxes; operations on them are handled by a separate component called the *object space*. The only way for the bytecode interpreter to gain actual knowledge about an object is to ask the object space for an its truth-value (i.e. whether the object is equivalent to True or False, information necessary for `if` statements). The object space, on the other hand, only knows about datatypes, not about executing Python code, for which it refers back to the interpreter.

High-level languages such as Python typically have a Foreign Function Interface (FFI) to interface to external C libraries. Because of the mismatch between the high-level language and C, FFIs are often clumsy to use. RPython's more static nature and lower-level types make it a better fit: consequently, PyPy and Converge mostly interface to C libraries in RPython.

Libraries which do not need to interface to external C libraries are more interesting in an RPython VM. Traditional VMs such as CPython implement as much functionality in C as is practical, often migrating libraries from the native language to C over time. The speed advantages of doing so are often huge, and such language communities develop careful conventions about what calculations should be done via library calls to take advantage of this. An important goal of RPython VMs is to significantly reduce the need to write and use builtin modules for performance reasons.

6. Optimising an RPython VM

Optimising an RPython VM means concentrating on the two execution modes: optimising the interpreter for faster interpretation speed; and rewriting the interpreter to produce traces which can be better optimised by the JIT. The former is largely similar to the challenges faced by other interpreters, so we dwell little on it; the latter is more unique to RPython VMs and what we concentrate on in this section.

From the perspective of an RPython VM author, many standard optimisations 'fall out of the hat'. Built-in datatypes such as integers, floats, and (to an extent) strings are naturally optimised by RPython's allocation removal techniques [15].

What an RPython VM author needs to concentrate on are the commonly used building blocks that are specific to the language being implemented. In the case of Converge and PyPy, the three common pinch points are instances (objects), classes, and modules[2]. As highly dynamic languages, Converge and Python programs can change and inspect run-time behaviour in arbitrary ways. However, most programs restrict such changes to small portions. Both Converge and PyPy therefore aim to make the common case of non-reflective access as fast as possible. Conversely, when a program uses the language's more dynamic features (introspection, self-modification, intercession [16]), execution falls back on slower, more general code.

In this section we give an overview of how language-specific building blocks can be optimised; many of the techniques described will be applicable to the different building blocks found in other languages[3]. In general, the Converge VM implements the 'easy win' optimisations, while PyPy optimises a much wider class of programs. Both experiences are useful: Converge shows how significant optimisations are possible with little effort, while PyPy shows how RPython VMs can optimise seemingly resistant programs.

6.1. General RPython JIT optimisation techniques

The techniques described in this section are more finely-tuned variants of the techniques described in [18]. The general aim is to produce small traces which can then be further shrunk by RPython's trace optimiser. The overall strategy is to expose, through rewriting the interpreter, the parts

[2]Informally, in both Converge and Python, a 'library' is a collection of modules.

[3]For container types (e.g. lists and hash maps) optimisation, see Diekmann [17].

which can be made constant in traces based on that code, and which can thus be optimised away, leaving only simple guards in their place. The tactics used to achieve this involve either using RPython-level annotations (i.e. promoting values and eliding functions) or rewriting the interpreter to use more trace-friendly code (e.g. moving from arbitrarily sized arrays to fixed-size lists). We now give a brief explanation of each.

6.1.1. Promoting values

In the context of a specific trace, it is often reasonable to assume that certain pieces of information are constant. The trace optimiser can be informed of this likelihood by *promoting* a value. For the small cost of inserting a guard at the first point of use, all subsequent calculations based on that constant can be constant-folded or removed from the trace. Note that constants are not known at compile-time: they are run-time values that are only constant for one particular trace. An important example of this is the concrete type of an object. Even in dynamically typed languages, most variables are only ever assigned values from a small set of types. Promoting the type of an object allows calculations to be specialised on that type. Because there is a very high likelihood that only a single type will be used at a given program point, the corresponding guard fails rarely.

6.1.2. Elidable functions

Similarly to promoting a value, functions can be annotated as being *elidable*. These are similar, though not identical, to the concept of pure functions. In short, an elidable function guarantees that, given the same inputs, it always returns the same outputs. In contrast to pure functions, elidable functions may have side effects, such as caching, providing they respect the overall relation between inputs and outputs. When a call to such a function is encountered in a trace, its body thus need not be executed when the input values match those previously encountered.

6.1.3. Using trace optimiser friendly code

Often two seemingly similar techniques can yield surprisingly different results in the context of the trace optimiser: one might frustrate the optimiser; another may allow it do its job well. As a concrete example, we look at the most common collections datatype: lists.

Arbitrarily growable lists cause the trace optimiser something of a headache. Every `append` (or similar) operation requires a check to see if the list has enough space left; if not, it must be resized, and possibly moved elsewhere in memory. Because of this, the trace optimiser then struggles to optimise much of the code which touches lists, as it is unable to definitively prove useful properties of the list over the lifetime of a trace. In contrast, fixed sized arrays have no such problems, and many accesses can be wholly optimised away. This is really another example of exposing constant information (in this case, the size of a list) to the trace optimiser. The difference between the two seemingly similar techniques can be huge.

The single biggest improvement in performance in the Converge VM was moving from a single global resizable list to a per-function frame fixed size stack. This necessitated modifying the compiler to statically calculate the maximum stack space a function requires at run-time (roughly 1 man day's work), and creating a fixed-size list of that size in each function frame.

Arbitrarily sized lists which are not randomly accessed need not be stored contiguously at all. A simple example of this is function frames; one needs to be able to access all of these to print out backtraces. In an old version of the Converge VM, these were stored in an arbitrarily sized list; clearly, such a list can not be replaced by a fixed size array as we have no idea in advance how deep functions will recurse. However, by having each function frame store a pointer to its parent (i.e. a singly linked list), the explicit list disappears entirely, and the trace optimisers life becomes much easier.

6.2. Optimising Instances

Both Converge and Python allow users to define classes and create instances (i.e. new objects) from them. Most programs of more than a few lines will creates such instances.

The basic semantics are similar in Converge and Python. Every instance records the class it instantiates; both languages allow this to be changed at run-time (in Python by writing to the `__class__` slot, in Converge to `instance_of`). Instances can also store an arbitrary number of slots (key/value pairs), which can vary on a per-instance basis (i.e. unlike many other OO languages, a class does not precisely define the 'shape' of its instances). In essence, instances behave like dictionaries mapping slot names (as strings) to values, while classes define the shared behaviour between instances. In this sense, both languages behave more like prototype-based languages such as Self than class-based languages such as Smalltalk.

Both Converge and PyPy optimise the common case of directly accessing slots in instances using *maps* (a concept originally from Self [19]; the technical

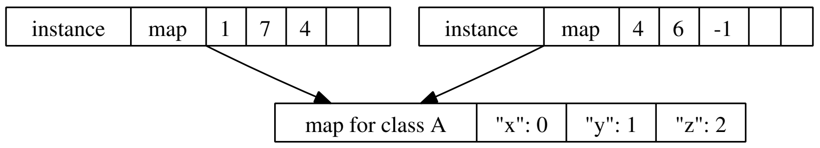

Figure 2: Two Instances of Class A Sharing the Same Map

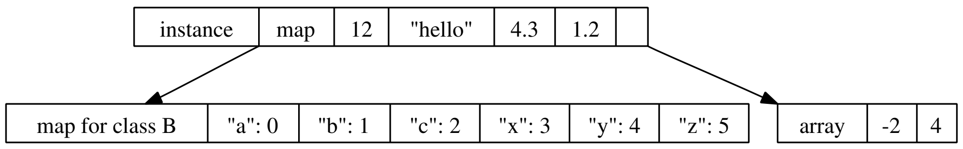

Figure 3: An Instance of Class B with Six Slots

details of PyPy's approach to maps are explained in [18]). Although instances of the same type can vary substantially, in practise the 'shape' of an instance is highly correlated with its class. Since the two rarely vary independently, PyPy stores the references to the class of an instance in its map, not directly in the object, saving a promotion in the process. Since promotions turn into guards in a trace, this produces smaller traces[4]. It also has the benefit of making objects one word smaller.

Although performance is PyPy's most obvious goal, it also attempts to save memory when that is not in direct conflict with performance. One example of this is PyPy's compact representation of instances. Observance of real systems showed that most objects have 5 or fewer slots. PyPy therefore preallocates space for 5 slots, freeing it from the need to allocate a arbitrarily sized list to store slots in most cases (which, when all its parts are taken account of, is up to 40% more memory needed to store 5 slots). Only when more than 5 slots are assigned to an instance is an arbitrarily sized list created and referenced from the object.

Figure 2 shows PyPy's layout scheme for two instances of class `A`, each instance using the same additional slot names. Since the instances have only three slots, the content of the slots can be stored in the free slots. Figure 3 shows an instance with six slots. Two of the fields have to be stored in an extra array allocated for that use. Note how the last field of the instance is

[4]Of course, many trace operations might be later optimised away; in this case, the guard resulting from the second promote would be unlikely to be so optimised.

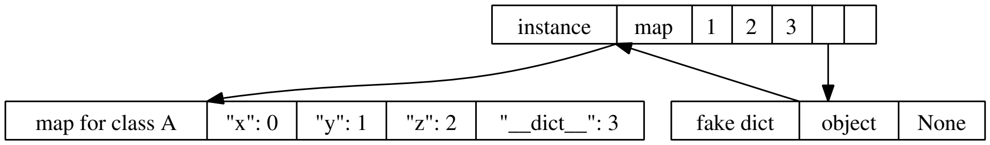

Figure 4: An Instance Implemented with a Map, and its Dictionary

used for that indirection.

6.2.1. Python's additional instance semantics

Python's instance model has a number of complexities over Converge's, which PyPy fully supports. These complexities are interesting because they show how interpreters can gradually allow performance to tail off as rarer, more dynamic, features are used.

The fundamental issue derives from an implementation decision in CPython: every instance has a reference, via the `__dict__` slot, to a dictionary that stores all the instance's slots. This dictionary can be replaced by writing to the `__dict__` slot, changing all the instance's slots. This implementation decision is costly in terms of memory, as dictionaries are not small data structures, and seems to defeat many reasonable optimisations.

A simple solution would be to use maps for normal accesses, but switch to storing an instance's slots in a dictionary stored in `__dict__` as soon as that slot is accessed. Doing so would mean that any reflective access of the dictionary would substantially slow down subsequent use of that instance. Since the dictionary is mostly used for reading and writing slots, this would slow down many real programs. Therefore, in PyPy, requesting an instance's dictionary returns a fake dictionary. This is indistinguishable from a real dictionary, and transparently redirects all reads and write to keys and values to the underlying instance. In other words, performance for normal accesses remains as fast as the standard case.

Figure 4 shows an instance, its map, and the fake dictionary that redirects all accesses back to the instance. Note that the instance needs to keep a reference to the dictionary once it has been requested in order to ensure that the expected object identity invariants are maintained.

However, when the programmer uses more of Python's dynamic features – in particular, writing a new dictionary to the `__dict__` slot – even this tactic is no longer viable. In such cases, PyPy stops using maps for the instance and stores its instances in a real dictionary (as shown in Figure 5). Fortunately

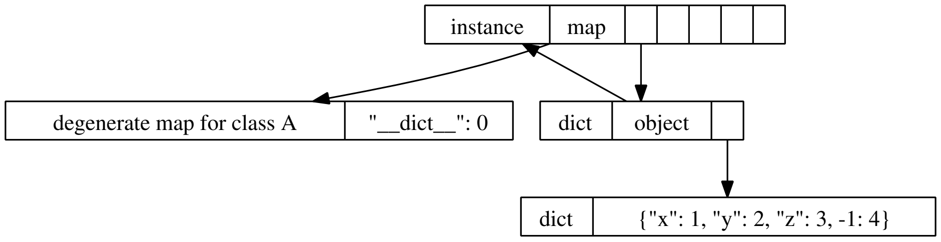

Figure 5: An Instance that has its Slots Stored in a Dictionary

such uses are rare, so few programs suffer the consequent slowdowns.

6.3. Optimising Classes

Both Python and Converge instances store only the information which varies from the class they instantiated. Typically this means that instances store dynamic information (ints, strings, user classes etc.) while classes store static information (typically functions). Accessing fields in classes is thus as common an operation as accessing slots in instances. Both PyPy and Converge aim to make non-reflective method lookup as fast as possible.

Looking up a method in a class necessitates, conceptually, traversing its inheritance hierarchy (note that both Converge and Python support multiple inheritance; Python uses the C3 algorithm, which means potentially looking at all its base classes during every method lookup). Since both languages allow classes to change dynamically, method lookup is a seemingly expensive operation.

The technique both languages use is to *version* classes. Every change to a class (e.g. adding or editing a field) changes its version. For any given version of a class, all of its fields are thus constant, and accesses to that class can be promoted (based on both the class *and* the version) and elided away. Because of inheritance, classes can not be versioned in isolation: for example, if a field is added to a class, then instances of its subclasses should gain that field too. Thus, as well as changing the version of a class when it is edited, we must change the versions of each of its subclasses. Since storing a normal reference to subclasses would prevent the latter ever being garbage collected, both Converge and PyPy classes store weak references (i.e. references that do not keep their target object 'alive') to their subclasses.

This technique makes looking up a field in a class extremely quick for the common case. The JIT optimises field lookups to a single guard which need only check that one class's version; if the check succeeds, the correct result is

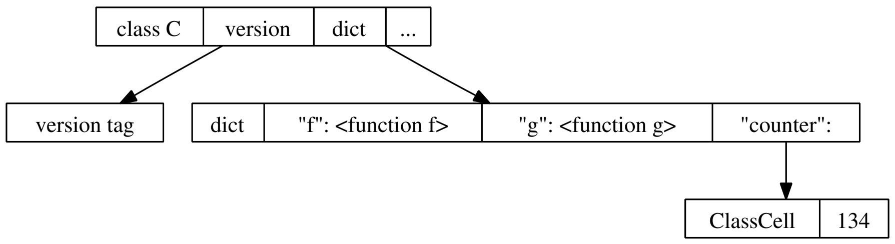

Figure 6: Class C with two Methods and a Counter

already known and inserted. Since versions can change an unbounded number of times, the seemingly obvious technique of a monotonically incrementing integer is dangerous: the integer could then overflow and two versions that were intended to be different could appear to be the same version, leading to unexpected behaviour. However, we only need compare whether one version is different than another, not whether one version is newer or older than another. Both PyPy and Converge therefore instantiate blank objects of an arbitrary class to stand in for versions: the RPython memory system implicitly guarantees that two different objects will compare differently, providing exactly the necessary guarantees, without any possibility of overflow.

However, as presented above, performance will suffer for the rarer case where fields in a class are frequently changed. One idiom which causes fields to change is a class which stores a counter so that it can give every instance a unique number. Every change thus requires changing the class and its subclass's versions; worse, trace guards are invalidated and new traces begun.

PyPy therefore adds one technique to the above (which Converge does not currently do). When a class field is changed for the first time, an extra level of indirection is introduced: the class no longer stores the field's value directly, but stores a reference to an small intermediate object that contains the value. When that particular field is changed subsequently, only the content of that object is changed, not the class as a whole: the class's version thus need not be changed. After the first time, writing to such a field thus causes relatively little slowdown, while reading from it needs an extra memory read (including when accessed via subclasses). While slightly less efficient, this gives a reasonable balance between fast general performance and reasonable performance in the rarer case.

Figure 6 shows a class with two methods `f` and `g` and a `counter` field. The counter is stored via an indirection to a `ClassCell`, meaning that changing it does not update the version of the class. On reading the `counter` slot, an extra pointer dereference is needed.

6.4. Optimising Modules

Modules are conceptually similar to classes (minus the complication of inheritance), and PyPy uses versioning technique similarly to classes. Converge uses a more naive scheme, to maintain simplicity in the compiler and VM. Top-level Converge module scopes are simply closures and within a module, they can be assigned to as normal; synchronising their mutation within and without the module would be somewhat difficult. Furthermore, tracking the number of assignments and adding indirection would be another complication. Converge modules thus use maps (which are promotable, and calculations on them easily elided) to map module lookups and assignments to an offset in a closure (which is a fixed size array). In practise, all reads and writes to Converge modules act like indirected accesses in PyPy. This gives reasonable (though not stellar) performance with little effort.

6.5. Discussion

With the optimisations described in this sections so far, instances, classes, and modules perform well in both the Python and the Converge VM. Instances are stored almost as compactly in memory as HotSpot, with equally efficient attribute access times, despite retaining sufficient information to implement highly dynamic languages. Classes are highly optimised for the common case (an inheritance hierarchy where methods in classes are not changed). In PyPy, module globals have most of their lookup overhead removed; in Converge, they are less efficient, but still adequately fast.

These optimizations exemplify how RPython VM authors need to consider which usage patterns are the most important (i.e. frequent) and therefore should be made as efficient as possible. They must then (re)arrange the interpreter and data structures so that, in conjunction with the trace optimiser, small traces with little code and few guards are produced. There is, of course, a tension between making common cases fast while not making less common cases unusably slow. VM authors need to understand their languages and intended use cases well. However, as often the case with performance issues, it is not realistic to do so purely intellectually: real programs must be analysed to determine which cases need to be focused on. Different benchmarks (synthetic or not) can change the perception of the most important areas substantially, and must be carefully chosen.

As this suggests, it is impossible to design a perfectly optimal interpreter up-front. Analysing the traces produced by real programs often shows places where they can be improved. Each pinch-point identified in the interpreter can be addressed either by adding hints for the JIT, or by rewriting the interpreter. Doing so is often not a trivial task, particularly as the interpreter becomes larger and more complex. It requires careful thought about the goals of the optimization, the trade-offs involved (including to code readability), and how to reach these goals.

Thus, while a basic meta-tracing JIT comes 'for free', an optimised one is no small task. That said, nearly all optimisations are understandable at the level of the interpreter itself: one need never look within the JIT compiler itself. The interpreter thus still expresses the language semantics correctly, albeit somewhat strangely when optimisations require changing its structure, and many optimisations improve the performance of the language interpreter as well as the resulting JIT. For example maps are a memory optimization if only an interpreter is used, and version tags can be used for a method cache within a purely interpreted system.

While promoting and eliding are direct features of the JIT, version tags are an idiom of use. This can understate their importance: they are a powerful way to constant-fold arbitrary functions on large data structures. The versions need to be updated carefully every time the result of a function on a structure can change. Therefore this is technique is only applicable on data structures which change slowly or which (as PyPy's approach to optimising classes in Section 6.3 showed) can be made to change slowly.

We believe that the manual rewriting of parts of the interpreter is a key part of the meta-tracing approach. Many of the optimisations rely on in-depth knowledge of the language the interpreter implements. The rewrites expose not only properties of the language semantics (which are already present in the interpreter) but also expectations about patterns of language use (which are not). While an 'optimally smart' meta-tracing compiler might be able to deduce some optimisations, most rely on the wider context which only a human can bring to the table.

7. Performance

In this section, we give a comparison of the performance of RPython against two extremes: hand-crafted high-performance VMs (HotSpot and

Language implementation	Underlying VM	Language version
HotSpot (1.7.0_147)	-	Java (1.7)
Lua (5.1.5)	-	Lua (5.1)
LuaJIT2 (git #5dbb6671a3)	-	Lua (5.1)
Converge1 (git #9084f0cdaf)	-	Converge (1.2)
Converge2 (git #e44800ec7c)	-	Converge (1.2)
CPython (2.7.2)	-	Python (2.7.2)
Jython (2.5.2)	HotSpot 1.7.0_147	Python (2.5)
PyPy (1.8)	-	Python (2.7.2)
PyPy–nonopt (1.8*)	-	Python (2.7.2)

Table 1: The language implementations we compare.

LuaJIT) and hand-crafted low-performance VMs (CPython, Lua, and Converge 1). In order to do this, we need programs which: run on all VMs; on the languages those VMs best support (Python, Converge, Lua, and Java); have implementations that are comparable in each language. Synthetic benchmarks are the only plausible candidates for cross-language comparison. With a reminder to readers of the inevitable limitations of synthetic benchmarks – they can easily be 'gamed' by language implementers and are often not representative of real workloads – we explain the systems under test, the methodology we use to measure performance, and the experimental results.

7.1. Systems under tests

The benchmarks we use are: Dhrystone (a venerable integer benchmark [20], and almost certainly the most widely ported cross-language benchmark); Richards[5] (which models task dispatch in an operating system); and Fannkuch-redux[6] (which counts permutations [21]). Dhrystone is included for its ubiquity; Richards for its relative real-worldism; and Fannkuch-redux for its exercising of built-in datatypes.

Table 1 shows the language implementations we compare, with detailed version information to ensure repeatability. LuaJIT2 and Converge2 (the RPython-based VM) have not, at the time of the writing, had formal releases so we give the specific git hash revision our experiments are based upon. Converge1 (the old C VM) is the Converge 1.2 VM with the minimal number

[5]`http://www.cl.cam.ac.uk/~mr10/Bench.html`

[6]`http://shootout.alioth.debian.org/u32/performance.php?test=fannkuchredux`

Benchmark	Short	Long
Dhrystone	50000	5000000
Fannkuch-redux	10	11
Richards	10	100

Table 2: Input size for short and long versions of the benchmarks.

of functions (related to integers and strings) added to allow the benchmarks to run. PyPy–nonopt is a PyPy variant without the major optimisations of Section 6 (the flags used to obtain this can be found in our repeatable build system), allowing us to explain the effect of those optimisations.

Our experimental system is almost fully automated to enable repeatability: it automatically downloads and builds the correct versions of the VMs (except for HotSpot), and then runs the experiments. We encourage interested readers to download it[7] and repeat the experiments.

7.2. Methodology

A fundamental problem when measuring JIT-based systems is whether to include warm-up time or not. JIT implementers often argue that warm-up times are irrelevant for long-running processes, and should be discounted. Others argue that many processes run for short time periods, and that warm-up times must be taken into account. We see merit in both arguments and therefore report two figures for each benchmark: *short*, where the benchmark has a low input size, and where warm-up times can play a significant part; and *long*, where a higher input size tends to dominate warm-up times. Table 2 shows the numbers involved.

We ran each version of the benchmark 30 times 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 3.0.0. We report the average wall time and confidence intervals with 95% confidence levels.

7.3. Experimental results

Table 3 shows the results from our experiment. Several things are worthy of note.

[7] `http://tratt.net/laurie/research/publications/files/metatracing_vms/`

	Dhrystone short (s)	Dhrystone long (s)	Richards short (s)	Richards long (s)	Fannkuch short (s)	Fannkuch long (s)
HotSpot	0.150 ± 0.018	0.476 ± 0.012	0.157 ± 0.010	0.271 ± 0.008	0.578 ± 0.025	6.215 ± 0.061
Lua	0.361 ± 0.008	36.558 ± 0.766	0.596 ± 0.016	12.033 ± 0.267	13.619 ± 0.286	177.926 ± 4.588
LuaJit	0.020 ± 0.004	2.085 ± 0.100	0.080 ± 0.000	1.429 ± 0.016	0.530 ± 0.000	6.945 ± 0.708
Converge1	3.709 ± 0.078	375.672 ± 8.669	18.246 ± 0.427	185.150 ± 4.716	†	†
Converge2	0.258 ± 0.008	5.859 ± 0.082	1.606 ± 0.010	8.794 ± 0.047	6.415 ± 0.155	79.714 ± 1.570
CPython	0.677 ± 0.012	65.621 ± 1.045	2.806 ± 0.022	27.976 ± 0.465	15.712 ± 0.860	195.150 ± 4.996
Jython	4.166 ± 0.478	60.006 ± 1.621	5.673 ± 0.145	28.672 ± 0.670	16.486 ± 1.147	167.376 ± 14.492
PyPy–nonopt	0.253 ± 0.012	12.777 ± 0.163	3.175 ± 0.025	3.183 ± 0.027	3.382 ± 0.069	41.246 ± 0.590
PyPy	0.150 ± 0.004	3.260 ± 0.076	0.598 ± 0.010	1.221 ± 0.016	3.242 ± 0.043	40.023 ± 0.986

Table 3: Benchmark Results.

The interpreter-only VMs (CPython, Converge1, and Lua) show similar, predictable (and often linear) slowdowns as the benchmarks lengthen. Lua is faster than CPython, which is faster than Converge1.

Jython, though not an interpreter but a compiler which creates Java bytecode, is slower on short benchmarks than CPython, and slightly faster on some long benchmarks. This is perhaps surprising, as it is using HotSpot as the underlying JIT—and HotSpot, on its own, is nearly always the fastest VM (particularly when it has warmed up). This underlines the 'semantic mismatch' problem we outlined on page 2.

Of the other JITted VMs, LuaJIT clearly outperforms PyPy and Converge2. This is particularly noticeable on the fast benchmark runs, which shows that RPython JITs warm-up rather sluggishly (see Section 8). Converge2 is dramatically faster than its non-JITted predecessor Converge1 (and, indeed, the interpreter-only VMs Lua and CPython), but can not compete with the more carefully tuned PyPy. Converge1 has a memory corruption bug which shows up in the Fannkuch-redux benchmark (and which had never been noticed before). Converge2 has no such problems, relying on RPython's automatic memory management.

PyPy–nonopt shows clearly the utility of the optimizations of Section 6—it is three times slower than standard PyPy. Dhrystone benefits from the the optimizations because it uses many global variables; Richards because it's written in a strongly object-orientated style. Fannkuch-redux sees fewer advantages from the optimisations, because it mostly does arithmetic and list manipulation.

8. Issues

As Section 7 shows, RPython VMs typically exceed hand-written interpreters in performance, even when less effort has been put into them. However, it would be foolish to pretend that RPython VMs are without issues. In this section, we enumerate the issues specific to RPython, and those common to all tracing approaches, as well as giving suggestions for possible solutions.

The most obvious problem with RPython VMs is time it takes to 'warm-up' the JIT (i.e. for all of the relevant hotspots in the code to have been traced and converted to machine code). Though all JITs suffer from this problem, the warm-up penalty in RPython VMs is larger because the JIT is language independent. Rather than having a custom tracer, the tracing interpreter (see Page 6) is used to generate tracers. The tracing interpreter is, in effect, itself interpreted to produce traces. Thus, during tracing, RPython VMs pay a double interpretation overhead. This is then compounded by the fact that meta-tracing inevitably creates large traces, which are expensive to produce, analyse, and optimise.

Since tracing is an expensive activity, RPython VMs have to be be cautious when determining whether it is sensible to start tracing. In practise, this means that loops must be executed an unusually large number of times before tracing is started. Long-running processes therefore take even longer to 'warm-up' than might be expected, and short-running processes often derive little or no benefit from the JIT.

Solving this problem would involve replacing the automatically created tracing interpreter with an equivalent component that is able to produce traces more efficiently. A plausible approach would be to specialize the tracing component to the language being implemented, removing the double interpretation overhead. The RPython project did some earlier experiments with that approach, but no current implementation remains.

The RPython language itself is not without issues. Some – such as a relative lack of documentation – are likely to be solved with time, and are not important enough to mention in detail here. A more fundamental problem is the 'Python' part of RPython. First, RPython is statically typed, but types can not be directly expressed in the language: they are instead inferred, with corresponding problems when inference goes awry. Second, RPython uses Python as its compile-time meta-programming language (roughly speaking, the language it uses to generate code at compile-time). The RPython translator loads in a normal Python file and executes it for as long as it chooses. Once that has finished, the translator expects to be given a reference to the VM's entry point, whence translation occurs. Everything referenceable from the entry point must be 'RPython enough' to be translatable; things not reachable are ignored (and may use arbitrary Python features). Compile-time meta-programming is vital for software that needs to be customisable and portable. However, it means that most VM files are in fact mixed Python and RPython programs. This mixing and matching of two similar, but distinct languages, is often confusing. Furthermore, it makes it difficult to translate RPython VMs in a modular fashion. Currently translation is 'whole program', and must be done for every single change. Large systems such as PyPy can take 45-60 minutes when a JIT is generated.

We suspect that future RPython-esque systems will choose a language with explicit static typing, and a clearly delineated compile-time meta-programming phase. For the former, a Java-esque language is likely to be sufficient; for the latter, a Converge-esque approach may yield good results.

A problem common to all tracing JITs (including those of RPython VMs) is that they give uneven performance improvements. Some programs run faster than method-based JITs while some run substantially slower. Programs in the latter category are invariably those which change the control flow paths they follow frequently. This causes guards to fail more often, and extra traces to be triggered. Compilers are a classic example of such programs (AST walkers appear, to a tracing JIT, to take different paths almost at random). The tracing JITs overhead can outweigh its benefits unless such programs run for a very long time and all the common paths are traced.

From the point of view of RPython VMs, this problem could only be solved by moving beyond the tracing paradigm. However, it is not clear how this might be done. First, tracing is a pragmatic way of getting reasonably good results for a wide variety of languages: other approaches are much harder to control and 'tune'. Second, as RPython shows, tracing is particularly amenable to 'meta' approaches: it would be harder to automatically create a method-based JIT, for example.

9. Related work

The techniques described in Section 6 are variations of well-known approaches. Maps were invented by the SELF project [2] to implement prototypes efficiently. Class versions can be related to the invalidation of method caches when a new method is compiled in a Smalltalk system [22].

An early attempt at making a generic environment for implementing dynamic languages was done also in the context of the SELF project [23]. That work builds a Smalltalk and a Java implementation on top of the SELF VM by compiling Java and Smalltalk bytecode to SELF bytecode, yielding good performance results due to the quality of the SELF VM. However, the SELF VM offers no way to customize what the JIT does with the bytecode for methods of another language, so it's not clear whether the approach can work for languages that are further from the execution model of SELF.

The only other meta-tracing system we are aware of is SPUR [24], a tracing JIT for CIL bytecode, which can be used as a meta-tracer for languages implemented in C#. The sole paper on SPUR uses meta-tracing to implement a JavaScript VM (we are not aware of it being applied to other languages). The JavaScript interpreter is carefully structured so that the traces that SPUR produces for common object operations are efficient, yielding excellent performance results. This is similar in intent to RPython, though SPUR works at the C# bytecode level, and has fewer ways to annotate the interpreter to produce efficient traces.

10. Conclusions

By looking at two different RPython VMs, we hope the reader has gained an understanding of the power of meta-tracing and its performance characteristics. We also detailed general optimisation techniques for meta-tracing VMs (as embodied in the PyPy and Converge VMs) that are likely, directly or indirectly, to aid future authors of meta-tracing VMs. We believe that further research into this area is likely to lead to increasingly impressive results which will narrow the performance gap with fully hand-crafted created JITs. For those prepared to pay the high manpower costs, hand-crafted JITs will always retain a performance edge; as this paper has demonstrated, language implementations can now perform at reasonable performance levels with surprisingly little effort.

Acknowledgements: We thank David Schneider and Samuele Pedroni for insightful comments on a draft of the paper. Fabio Mascarenhas kindly placed the Lua Richards benchmark into the public domain. We thank the PyPy and RPython community for their continuous support and work: Armin Rigo, Maciej Fijałkowski, Alex Gaynor, and countless others. Any remaining errors and infelicities are our own.

References

[1] L. Tratt, Dynamically typed languages, Advances in Computers 77 (2009) 149–184.

[2] C. Chambers, D. Ungar, Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language, in: Proc. PLDI, ACM, 1989.

[3] M. Paleczny, C. Vick, C. Click, The Java HotSpot server compiler, in: Proc. JVM Research and Technology Symposium, USENIX, 2001.

[4] B. P. Serpette, M. Serrano, Compiling scheme to JVM bytecode: a performance study, SIGPLAN Not. 37 (2002) 259–270.

[5] J. R. Rose, Bytecodes meet combinators: invokedynamic on the JVM, Proc. VMIL (2009).

[6] D. Ancona, M. Ancona, A. Cuni, N. D. Matsakis, RPython: a step towards reconciling dynamically and statically typed OO languages, in: DLS, ACM, 2007.

[7] A. Rigo, S. Pedroni, PyPy's approach to virtual machine construction, in: DLS, ACM, Portland, Oregon, USA, 2006.

[8] L. Tratt, Compile-time meta-programming in a dynamically typed OO language, in: Proc. DLS, ACM, 2005, pp. 49–64.

[9] L. Tratt, Experiences with an Icon-like expression evaluation system, in: Proc. DLS, ACM, 2010, pp. 73–80.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: the story of squeak, a practical smalltalk written in itself, in: Proc. OOPSLA, ACM, 1997, p. 318–326.

[11] R. A. Kelsey, Pre-Scheme: a scheme dialect for systems programming, 1997.

[12] C. F. Bolz, A. Cuni, M. Fijałkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: ICOOOLPS, ACM, 2009, pp. 18–25.

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

[14] A. Gal, C. W. Probst, M. Franz, HotpathVM: an effective JIT compiler for resource-constrained devices, in: VEE, ACM, 2006.

[15] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, A. Rigo, Allocation removal by partial evaluation in a tracing JIT, in: Proc. PEPM, pp. 43–52.

[16] G. Bracha, D. Ungar, Mirrors: design principles for meta-level facilities of object-oriented programming languages, in: Proc. OOPSLA, ACM, 2004, pp. 331–344.

[17] Lukas Diekmann, Memory Optimizations for Data Types in Dynamic Languages, Master thesis, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, 2012.

[18] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, A. Rigo, Runtime feedback in a meta-tracing JIT for efficient dynamic languages, in: Proc. ICOOOLPS, ACM, 2011, p. 9:1–9:8.

[19] C. Chambers, D. Ungar, E. Lee, An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes, in: OOPSLA, volume 24, ACM, 1989.

[20] R. P. Weicker, Dhrystone benchmark (Ada version 2): rationale and measurements rules, Ada Lett. IX (1989) 60–62.

[21] K. R. Anderson, D. Rettig, Performing Lisp analysis of the FANNKUCH benchmark, SIGPLAN Lisp Pointers VII (1994) 2–12.

[22] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: POPL, ACM, Salt Lake City, Utah, 1984.

[23] M. Wolczko, O. Agesen, D. Ungar, Towards a universal implementation substrate for Object-Oriented languages, in: Workshop on Simplicity, Performance, and Portability in Virtual Machine Design.

[24] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, H. Venter, SPUR: a trace-based JIT compiler for CIL, in: Proc. OOPSLA, pp. 708–725.