
**High performance implementation of Python for
CLI/.NET with JIT compiler generation for dynamic
languages.**

by

Antonio Cuni

Theses Series

DISI-TH-2010-05

DISI, Università di Genova

v. Dodecaneso 35, 16146 Genova, Italy

<http://www.disi.unige.it/>

Università degli Studi di Genova

**Dipartimento di Informatica e
Scienze dell'Informazione**

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

**High performance implementation of Python
for CLI/.NET with JIT compiler generation
for dynamic languages.**

by

Antonio Cuni

July, 2010

**Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova**

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
<http://www.disi.unige.it/>

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Antonio Cuni
DISI, Univ. di Genova
cuni@disi.unige.it

Date of submission: July 2010

Title: High performance implementation of Python for CLI/.NET with JIT compiler
generator for dynamic languages.

Advisor: Davide Ancona
DISI, Univ. di Genova
davide@disi.unige.it

Ext. Reviewers:
Michael Leuschel
STUPS Group, University of Düsseldorf
leuschel@cs.uni-duesseldorf.de

Martin von Löwis
Hasso-Plattner-Institut
martin.vonloewis@hpi.uni-potsdam.de

Abstract

Python is a highly flexible and open source scripting language which has significantly grown in popularity in the last few years. However, all the existing implementations prevent programmers from developing very efficient code. This thesis describes a new and more efficient implementation of the language, obtained by developing new techniques and adapting old ones which have not yet been applied to Python. Even though the specific implementation considered here targets the .NET framework, extreme care has been taken to develop a compiler easily portable to other platforms, and reusable for the implementation of other dynamic languages.

As a language, Python is very hard to implement efficiently: the presence of highly dynamic constructs makes static analysis of programs extremely difficult, thus preventing ahead of time (AOT) compilers to generate efficient target code.

CPython, the reference implementation of the language, is an interpreter and its performance is far from optimal. Jython and IronPython are two alternative implementations that target respectively the Java Virtual Machine and the .NET framework: differently from CPython, they are compilers which generate bytecode for the underlying virtual machine, but often their performance is not better than CPython.

An alternative solution to this problem, which has been already investigated for other object-oriented languages, is the implementation of a Just in Time (JIT) compiler. Unfortunately, writing a JIT compiler is far from being an easy task, and most of the times efficiency is gained at the cost of a dramatic loss of portability.

To achieve this, the PyPy project[pyp] has been exploited. PyPy consists of both a mature, fully compatible Python interpreter and a framework to transform it in various way. In particular, it provides a JIT compiler generator, which automatically turns the interpreter into a JIT compiler.

The contribution of this thesis includes both the work done in cooperation with the rest of the PyPy team for enhancing the PyPy JIT compiler generator, and for developing a back-end to target the .NET framework. In particular, implementing a JIT compiler for the .NET virtual machine rather than for a CPU has allowed the combination of two different JIT compilation layers: the PyPy one, which produces .NET bytecode, and the .NET one which produces executable machine code. This thesis demonstrates that this combination gives very good results in terms of performance.

Finally, adopting the PyPy approach turned out to be successful for ensuring portability and reuse. Indeed, it is relatively simple to implement new dynamic languages in PyPy and get JIT compilers almost for free, but it is also possible to develop new back-ends for the JIT generator, in order to target new platforms, e.g. the JVM.

To Nicolò and Veronica

*Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.*

*Consider your origin:
you were not born to live like brutes,
but to follow virtue and knowledge*

Divina Commedia,
Inferno canto XXVI, 118-120
(Dante Alighieri)

Acknowledgements

First, I wish to thank Davide Ancona for his support during the years of my PhD and his always useful suggestions during the writing of this thesis.

I am grateful to all the people I had as friends and colleagues, both at DISI in Genoa and in the PyPy Team: Marco, Giovanni, Elena, Walter, Paolo, Daniele, Armin, Carl Friedrich, Maciej, Samuele, Holger, Bea, Laura, Jacob, and all the people that I surely forgot: I hope you won't take it as a cliché when I say that spending time with you has really been a pleasure.

Table of Contents

Chapter 1	Introduction	6
1.1	Python implementations	6
1.1.1	Digression: Python 2 vs. Python 3	7
1.2	A gentle introduction to Just In Time compilation	8
1.3	Related work	9
1.4	Contributions of this thesis	10
1.5	Structure of the thesis	11
Chapter 2	The problem	12
2.1	Is Python intrinsically slow?	12
2.1.1	Interpretation overhead	13
2.1.2	Boxed arithmetic and automatic overflow handling	13
2.1.3	Dynamic dispatch of operations	14
2.1.4	Dynamic lookup of methods and attributes	14
2.1.5	The world can change under your feet	15
2.1.6	Extreme introspective and reflective capabilities	17
2.2	Interpreters vs. compilers: limits of static analysis for Python	18
2.3	Our solution: automatically generated JIT compiler	19
Chapter 3	Enter PyPy	20
3.1	What is PyPy?	20

3.2	Architecture overview	21
3.3	JIT compiler generator	22
3.4	Why choosing PyPy	23
Chapter 4 Characterization of the target platform		26
4.1	Dynamic loading of new bytecode	27
4.2	JIT layering	27
4.3	Immutable methods	28
4.4	Tail calls	29
4.4.1	Tail calls benchmark	29
4.5	Primitive types vs reference types	30
4.6	Delegates (or: object oriented function pointers)	31
Chapter 5 Tracing JITs in a nutshell		32
5.1	What is a tracing JIT compiler?	32
5.2	Phases of execution	33
5.3	A deeper look at traces	34
5.4	Example of tracing	35
5.5	Generalization of the trace and code generation	38
5.6	Linear traces vs. trace trees	39
Chapter 6 The PyPy JIT compiler generator		42
6.1	There is a time for everything	42
6.2	Architecture of the JIT compiler generator	43
6.3	Tracing the meta level	45
6.3.1	Applying a tracing JIT to an interpreter	45
6.3.2	Detecting user loops	47
6.3.3	Applying the hints	48
6.4	Loops, bridges and guards	51

6.5	Escape analysis and trace optimization	53
6.5.1	Escape analysis	54
6.5.2	Reentering a specialized loop	55
6.5.3	Forcing virtuals	56
Chapter 7 The CLI JIT backend		60
7.1	Teaching the JIT frontend Object Orientation	60
7.2	The interface between the JIT frontend and backend	61
7.2.1	Descriptors	62
7.2.2	Executing operations	62
7.2.3	Loops and bridges	63
7.2.4	Executing loops	64
7.3	Compiling loops	65
7.4	Code generation	66
7.5	Compiling inner bridges	68
7.5.1	Entry and exit bridges	70
Chapter 8 Benchmarks		72
8.1	Applying the JIT to the Python Interpreter	72
8.2	Making the interpreter JIT friendly	73
8.3	Methodology	74
8.4	Microbenchmarks	75
8.5	Middle-sized benchmarks	77
8.5.1	Differences between Mono and CLR	80
Chapter 9 Conclusion and Future Work		84
9.1	Contributions of this thesis	84
9.2	Future work	86

Chapter 1

Introduction

1.1 Python implementations

Python [pyt] is a highly flexible and open source scripting language which has significantly grown in popularity in the last few years. The reference implementation is called *CPython* and is written in C. The language is implemented by a compiler that translates Python source code into a very high level bytecode, and by a simple virtual machine that interprets the bytecode.

Since the beginning, one of CPython goals have been to keep the implementation simple to read, maintain and extend, in order for the language to be easier to mature and evolve. However, one drawback of this approach is that the final performance of the programs are not as good as they could be.

Although CPython is the reference implementation, it is not the sole. In particular, over the years four main different alternative implementations emerged: *Jython* [jyt], *IronPython* [iro], *PyPy* [pyp] and *Unladen Swallow* [unl]¹.

Jython is an implementation of Python written in Java and running on the Java Virtual Machine (JVM). Contrarily to CPython, it does not implement an interpreter but instead it compiles Python source code into JVM bytecode.

IronPython is the Jython equivalent for the .NET Framework [dot]. As Jython, it translates Python source code to bytecode for the CLI [Int06], which is the virtual machine at the heart of the .NET Framework. Due to the differences between the object model of Python and the native object model of the JVM or the CLI, both Jython and IronPython

¹These are not the only alternatives to CPython. In particular, there are forks that extend CPython to have new features such as Stackless Python and there are compilers for Python-like languages which aims at better performance. However, they are not relevant for the topic of this thesis.

suffer performance problems when they have to deal with some highly dynamic features of Python.

Since the language is continuously evolving, it is hard to keep all the alternative implementations synchronized with the reference one. Among the others, this is one of the problems which **PyPy** aims to solve. PyPy is an implementation of the language written in Python itself: the main idea is to write a high level specification of the interpreter in a restricted subset of Python (called RPython, for Restricted Python), to be translated into lower-level efficient executables for the C/POSIX environment, for the JVM and for the CLI, thus providing a platform independent implementation of Python which can be easily ported by adding the corresponding backends. Moreover, the other major goal of PyPy is to develop a Just In Time compiler for Python, in order to improve the performance of the languages. Finally, PyPy is much more than just an implementation of Python: indeed, it is a general framework to implement effectively and efficiently all kinds of dynamic languages for various target platforms.

Despite the fact that Jython and IronPython are compilers and not interpreters, their performance is not significantly better than CPython, and often they are even slower. This is partly due to the fact that Python is inherently hard to implement efficiently, and partly because neither the JVM nor the CLI have been designed to implement dynamic languages. The scope of this thesis is to demonstrate that it is possible to write very efficient implementations of Python, and more generally of dynamic languages, on top of these virtual machines, by porting the Just In Time compilation techniques of PyPy to the CLI and the JVM as well.

Finally, the last Python implementation that came out in chronological order is **Unladen Swallow**: differently from the others, Unladen Swallow is not written from scratch but it is a fork of CPython to seek for better performance. As PyPy, Unladen Swallow implements a Just In Time compiler for Python: however, differently than Unladen Swallow this thesis is centered on targeting virtual machines such as the CLI, so the problems we had to face are different.

1.1.1 Digression: Python 2 vs. Python 3

Since the beginning each new version of Python has always been backward compatible with the preceding ones. This rule has been broken with the advent of Python 3.0, which has been intentionally made incompatible to remove old, unwanted features and introduce new ones which could not be added in a backward compatible manner. At the moment of writing, CPython is the only available implementation of Python 3, while all the other alternative implementations are still targeting Python 2.x.

For this reason, in this thesis we talk about Python 2.x: however, all the concepts apply

equally well also to Python 3, and to dynamic languages in general.

1.2 A gentle introduction to Just In Time compilation

As [AFG⁺05] nicely illustrates, over the years the problem of making dynamic languages faster has been tackled in a number of different ways, which can be divided into two large categories:

1. writing fast interpreters
2. writing optimizing Just In Time (JIT) compilers

The key idea behind fast interpreters is to reduce at minimum the cost of interpretation overhead. The various techniques to meet this goal are greatly summarized by [EG03]: however, they are effective only for those languages for which the cost of interpretation dominates the total execution time. As we will see in Section 2.1.1, this is not the case for Python.

On the other hand, JIT compilers have proved to be very effective at optimizing different kinds of overhead introduced by dynamic languages. The main characteristic of JIT compilers is that they generate machine code at runtime and only when needed (hence the name, *just in time*): usually the program is represented in form of a bytecode for a virtual machine, which is portable and architecture independent, to be translated to machine code.

Depending on the strategy, a JIT compiler can be either the only way to execute the bytecode or it can work side by side with an interpreter. In the latter case, usually the program is interpreted at first, then the JIT compiler optimizes only the *hotspots*, i.e. the parts of the program which are executed very often and thus are responsible for the majority of the time spent.

To underline better the differences between the various strategies, we introduce a bit of terminology:

- A **Batch** or **Ahead Of Time** (AOT) compiler translates the source code (or the bytecode) into machine code before the program starts.
- A **Just In Time** (JIT) compiler translates the source code (or, more often, the bytecode) into machine code immediately before executing it, possibly caching the result to avoid that the same code fragment is compiled twice. In the subsequent chapters, we will abbreviate the term JIT compiler with “JIT”.

- A **Static** compiler uses exclusively the information which can be derived from the source code or from the bytecode to perform the translation process.
- A **Dynamic** or **Adaptive** compiler uses also extra information collected at runtime: for example, the code paths which are taken most frequently, the dynamic types of the objects stored in a given variable, and so on.

Obviously, dynamic compilers have better chances to generate more efficient code, since they have more knowledge about the program than their static counterpart. Following this definition, it is also clear that while almost all AOT compilers are static, not all JIT compilers are dynamic.

1.3 Related work

Historically, the first important contribution to this research area consists in the implementation of the *SELF* language [H94], where several proposed novel techniques have been reused and enhanced to implement highly efficient virtual machines such as *HotSpot* for Java [PVC01].

Another interesting project for more recent languages is *Psyco* [Rig04]: it is an implementation of a dynamic JIT compiler for Python, which generalizes the techniques developed for SELF, in particular the concept of *Polymorphic Inline Cache* (PIC) [HCU91]. It gets very good results, as programs optimized by Psyco can run up to 60 times faster than CPython. However, Psyco has some drawbacks:

- While on the surface Python is a clean and simple language, under the hood it has a very complex semantics with many corner cases: thus, developing a compiler which keeps the exact same behaviour as the reference implementation is very hard and time consuming.
- For the same reason, each time the reference implementation evolves, e.g. by adding a new feature, the JIT compiler needs to be updated. Over the time, adding new features become more and more complex, as they may interact each other in various ways. For example, implementing support for *generators*²[SPH01] in Psyco took approximately four years, proving that this approach does not scale well as the language changes.

²In Python, *generators* are special functions that can be interrupted and resumed at specific points, returning a new value at each resume

- Because of the way it is designed, Psyco compiles all code that is executed from each of the starting points that are profiled, instead of compiling only the hotspots. As a consequence, it ends up using too much memory, limiting its uses.

The author of the thesis has already explored the implementation of a Psyco-style JIT compiler for the CLI in [CAR09] and [ABCR08]: however, the solutions based on Polymorphic Inline Caches rely on the fact that it is possible to modify the code at runtime, after it has already been executed. As Section 4.3 explains, this feature is not supported by the Object Oriented Virtual Machines like the CLI or the JVM and all the solutions to try to overcome this limitation are not particularly efficient.

Finally, there is one particular strategy for implementing dynamic compilers called *Tracing JIT compilation*: it was initially explored by the Dynamo project [BDB00] to dynamically optimize machine code at runtime. Its techniques were then successfully used to implement a JIT compiler for a Java VM [GPF06, GF06]. Subsequently these tracing JITs were discovered to be a relatively simple way to implement JIT compilers for dynamic languages [CBY⁺07]. The technique is now being used by both Mozilla's TraceMonkey JavaScript VM [GES⁺09] and has been tried for Adobe's Tamarin ActionScript VM [CSR⁺09].

Tracing JITs are built on the following basic assumptions:

- programs spend most of their runtime in loops
- several iterations of the same loop are likely to take similar code paths

The tracing JIT approach is based on the idea that machine code is generated only for the hot code paths of mostly executed loops, while the rest of the program is interpreted. The code for those mostly executed loops is highly optimized, including aggressive inlining. The difference with more traditional approaches, such as the one implemented by Java HotSpot, is the granularity of JIT compilation: HotSpot compiles whole methods at once, while Tracing JIT compilers compile loops, which can either be contained in a single method, or span over several ones in case of inlining. Chapter 5 explains in details how tracing JITs work.

1.4 Contributions of this thesis

This thesis explores and investigates new techniques to implement in an efficient and effective way dynamic languages on top of object oriented, statically typed virtual machines. In particular, it introduces the concept of *JIT layering* (see Section 4.2): the idea is that the host VM and its low level JIT compiler do not know enough about the dynamic language in

question to optimize it in a proper way. Instead, we add a higher layer of JIT compilation that is specific for the language and dynamically emits bytecode for the target VM that is further compiled to by the low level JIT.

Instead of writing a JIT compiler by hand, we adapt and use the PyPy JIT compiler generator (see Chapter 6), which automatically generates a JIT compiler from the source code of an interpreter for the given language. It is important to underline that the work done on the PyPy JIT compiler generator it is not solely of the author of the thesis but of the whole PyPy team. To make the PyPy JIT compiler generator working with the CLI virtual machine, we wrote the CLI JIT backend (see 7).

Finally, we apply the PyPy JIT compiler generator to the PyPy Python Interpreter to get a JIT compiler for Python targeting the CLI, and we measure its performance on a set of benchmarks against IronPython (see Chapter 8), the other mainstream and mature implementation of Python for the CLI.

1.5 Structure of the thesis

The rest of this thesis is structured as follows.

Chapter 2 provides an overview on the current implementations of Python and explains why implementing Python efficiently is so hard.

Chapter 3 introduces PyPy, which is the project which thesis is based on.

Chapter 4 presents the target platform, the CLI. It contains a description of the virtual machine, of its peculiarities, and a comparison with the JVM.

Chapter 5 explains how tracing JIT compilers work in general.

Chapter 6 explains more in detail the tracing JIT techniques employed by PyPy. In particular, it also explains how the JIT compiler is automatically generated from the source code of the interpreter.

Chapter 7 describes the functioning of CLI JIT backend of PyPy, and the problems that have been encountered during its development.

Chapter 8 explains how the JIT compiler generator has been applied to the PyPy Python Interpreter, and measures its performance when translated with the CLI JIT backend against IronPython.

In Chapter 9, we conclude the work by summarizing the contributions presented in this thesis, describing related works and anticipating future directions of research.

Chapter 2

The problem

2.1 Is Python intrinsically slow?

At the moment, there are four main implementations of Python as a language: CPython is the reference implementation, written in C; Jython is written in Java and targets the JVM; IronPython is written in C# and targets the CLI (i.e., the VM of .NET); finally, PyPy is written in Python itself and targets a number of different platforms, including C, JVM and .NET. In the following, we are discussing CPython, when not explicitly stated otherwise; most of concepts are equally applicable to the other implementations, though.

Running Python code is slow; depending on the benchmark used, Python programs can run up to 250 times slower than the corresponding programs written in C.

There are several reasons that make Python so slow; here is a rough attempt to list some of what we think are the most important ones:

1. Interpretation overhead
2. Boxed arithmetic and automatic overflow handling
3. Dynamic dispatch of operations
4. Dynamic lookup of methods and attributes
5. *“The world can change under your feet”*
6. Extreme introspective and reflective capabilities

2.1.1 Interpretation overhead

In CPython, Python programs are compiled into bytecode which is then interpreted by a virtual machine. The interpreter consists of a *main interpreter loop* that fetches the next instruction from the bytecode stream, decodes it and then jumps to the appropriate implementation: this is called *instruction dispatching*. Compared to emitting directly executable machine code, the extra work needed for instruction dispatching adds some overhead in terms of performance, which is called *interpretation overhead*.

The interpretation overhead strongly depends on the nature of the bytecode language: for languages whose bytecode instructions are fast to execute the time spent to do instruction dispatching is relatively higher than for languages whose instructions take more time.

For Python, it has been measured that the interpretation overhead causes roughly a 2.5x slowdown compared to executing native code[CPC⁺07].

Both Jython and IronPython do not suffer from this problem, since they emit bytecode for the underlying virtual machine (either JVM or CLI), which is then translated into native code by the Just In Time compiler.

2.1.2 Boxed arithmetic and automatic overflow handling

In Python all values, including numbers, are objects; the default type for signed integers is `int`, which represents native integers (i.e., numbers whose length is the same as a WORD, typically 32 or 64 bit).

Moreover, Python provides also a `long` type, representing arbitrary precision numbers; recent versions of Python take care of switching automatically to `long` whenever the result of an operations cannot fit into `int`.

The most natural way to implement these features is to allocate both values of type `int` and `long` on the heap, thus giving them the status of “objects” to switch from one to the other very easily.

Unfortunately, this incurs a severe performance penalty. Figure 2.1 shows a simple loop doing only arithmetic operations, without overflows. Benchmarks show that this code runs about 40 times slower than the corresponding program written in C and compiled by `gcc` 4.4.1 without optimizations. For more information on the benchmarking machine, see Section 8.3.

```
_____  
i = 0  
while i < 10000000:  
    i = i+1  
_____
```

Figure 2.1: *Simple loop*

However, most of the integer objects used in Python programs does not escape the function

in which they are used and never or rarely overflow, so they do not really need to be *effectively* represented as objects in the heap; a smart implementation could detect places where numbers can be stored “on the stack” and unboxed, and emit efficient machine code for those cases.

The same discussion applies equally well also to floating point numbers, with the difference that in this case we do not need to check for overflows at every operation.

2.1.3 Dynamic dispatch of operations

Most operations in Python are dynamically overloaded, i.e., they can be dynamically applied to values of different types. Consider for example the `+` operator: depending on the types of its arguments, it can either add two numbers, concatenate two strings, append two lists, or call some user-defined method.

Consider again the loop in Figure 2.1 and an excerpt of its bytecode, shown in Figure 2.2. The `BINARY_ADD` operation corresponds to the `i+1` expression.

Being dynamically typed, the virtual machine does not know in advance the types of the operands of `BINARY_ADD`, hence it has to check them at runtime to select the proper implementation. This could lead to poor performance, especially inside a loop: for the loop in question, the virtual machine always dispatches `BINARY_ADD` to the same implementation. In this example, it is very clear that an efficient implementation could detect that there are a fast and a slow path, and emit efficient code for the former.

```

# while i < 10000000
 9 LOAD_FAST          0 (i)
12 LOAD_CONST         2 (10000000)
15 COMPARE_OP         0 (<)
18 JUMP_IF_FALSE     14 (to 35)
21 POP_TOP

# i = i + 1
22 LOAD_FAST          0 (i)
25 LOAD_CONST         3 (1)
28 BINARY_ADD
29 STORE_FAST         0 (i)

# close the loop
32 JUMP_ABSOLUTE     9

```

Figure 2.2: Bytecode for the loop in Figure 2.1

2.1.4 Dynamic lookup of methods and attributes

Python is an attribute based language: the forms `obj.x` and `obj.x = y` are used to get and set the value of the attribute `x` of `obj`, respectively. In contrast to other object oriented language, message sending or method invocation is not a separate operation than attribute access: methods are simply a special kind of attribute that can be called.

```

1 # an empty class
2 class MyClass:
3     pass
4
5 # arguments passed to foo can be of any type
6 def foo(target, flag):
7     # the attribute x is set only if flag is True
8     if flag:
9         target.x = 42
10
11 obj = MyClass()
12 # if we pass False, obj would not have any attribute x
13 foo(obj, True)
14 print obj.x
15 print getattr(obj, "x")    # use a string for the attribute name

```

Figure 2.3: *Dynamic lookup of attributes*

The process to get the value of an attribute is called *attribute lookup*. The semantics of the attribute lookup differs depending on the dynamic type of the object in question: for example, if the object is an instance of a class the attribute will be first searched in the object, then in its class, then in its superclasses. Moreover, classes can override the default behavior by defining the special method `__getattr__` which is called whenever an attribute lookup occurs.

Since its semantics depends on the dynamic type of the objects, in Python the lookup of attributes is performed at run-time. Moreover, there are special built-in functions that allow to get and set attributes by specifying their name as a string. For example, the code in Figure 2.3 is a valid Python program that prints 42 twice. Note that the behavior of line 14 depends on the value of the flag passed to `foo`: if we passed `False`, an exception would be raised.

As usual, if on the one hand these features allow a great degree of flexibility, on the other hand they are very difficult to implement efficiently; since the set of methods and attributes is not statically known, it is impossible to use a fixed memory layout for objects, as it happens in the implementation of statically typed object-oriented languages based on the notion of virtual method tables. Instead, classes and objects are usually implemented using a *dictionary* (i.e. a hashtable) that maps attributes, represented as strings, to values, whilst methods and instance variables are typically managed differently in statically typed object oriented languages.

2.1.5 The world can change under your feet

In Python, the global state of the program can continuously and dynamically evolve: almost everything can change during the execution, including for example the definition of functions and classes, or the inheritance relationship between classes.

Consider, for example, the program in Figure 2.4: the function associated to the name `fn` changes at runtime, with the result that two subsequent invocations of the same name can lead to very different code paths.

The same principle applies to classes: we have already seen in section 2.1.4 that the attributes of an object can dynamically grow runtime, but things can change even deeper: the example in Figure 2.5 demonstrates how objects can change their class at runtime, even for unrelated classes. Note also that `my_pet` preserves its original attributes.

Thus, in Python it is usually unsafe to assume anything about the outside world, and we need to access every single class, function, method or attribute dynamically, because it might not be what used to be previously.

```
def fn():
    return 42
def hello():
    return 'Hello world!'
def change_the_world():
    global fn
    fn = hello

print fn() # 42
change_the_world()
print fn() # 'Hello world!'
```

Figure 2.4: *Changing the global state*

```
class Dog:
    def __init__(self):
        self.name = "Fido"
    def talk(self):
        print "%s: Arf! Arf!" % self.name

class Cat:
    def __init__(self):
        self.name = "Felix"
    def talk(self):
        print "%s: Meowww!" % self.name

my_pet = Dog()
my_pet.talk() # Fido: Arf! Arf!
my_pet.__class__ = Cat
my_pet.talk() # Fido: Meowww!
```

Figure 2.5: *Dynamically changing the class of an object*

2.1.6 Extreme introspective and reflective capabilities

Python offers a lot of ways to inspect and modify a running program. For example, you can find out the methods of a class, the attributes of an instance, the names and the values of all locals variables defined in the current scope, and so on.

Moreover, often it is also possible to modify this information: it is possible to add, remove or modify methods of a class, change the class of an object, etc. Since in CPython efficiency is not considered a compelling requirement, all these features are implemented in a straightforward way; however, it is very challenging to find alternative ways to implement them efficiently.

In particular, CPython allows access to all the frames that compose the current execution stack: each frame contains information such as the instruction pointer, the frame of the caller and the local variables; moreover, under some circumstances it is possible to mutate the value of the local variables of the callers, as shown by the example in Figure 2.6¹.

The function `sys.settrace()` is another example of feature that add overheads to the overall execution time of programs. It allows the programmer to install a *tracer* to monitor and possibly alter the execution of the program: the events notified to the tracer include entering/leaving a function, executing a new line of code, raising an exception. Figure 2.7 shows an example of using `sys.settrace`, and Figure 2.8 shows its output.

These functionalities are heavily used by some applications such as debuggers or testing frameworks, even though their abuse in “regular code” is strongly discouraged.

```
def fill_list(name):
    # get the caller frame object
    frame = sys._getframe().f_back

    # get the variable "name" in
    # the caller's context
    lst = frame.f_locals[name]
    lst.append(42)

def foo():
    mylist = []
    fill_list("mylist")
    print mylist    # prints [42]
```

Figure 2.6: *Introspection of the stack frame*

¹The official documentation says that `sys._getframe` is a CPython implementation details. However it is used by real world programs, so it is required to be a real world alternative to CPython.

<pre> 1 import sys 2 3 def test(n): 4 j = 0 5 for i in range(n): 6 j = j + i 7 return j 8 9 def tracer(frame, event, arg): 10 fname = frame.f_code.co_filename 11 lineno = frame.f_lineno 12 pos = '%s:%s' % (fname, lineno) 13 print pos, event 14 return tracer 15 16 sys.settrace(tracer) # install the tracer 17 test(2) # trace the call </pre>	<pre> settrace.py:3 call settrace.py:4 line settrace.py:5 line settrace.py:6 line settrace.py:5 line settrace.py:6 line settrace.py:5 line settrace.py:7 line settrace.py:7 return </pre>
---	---

Figure 2.7: *Example of using `sys.settrace`*

Figure 2.8: *Output*

2.2 Interpreters vs. compilers: limits of static analysis for Python

During the years, people often proposed to make Python faster by implementing a compiler, which should be supposedly much faster than an interpreter. However, if you carefully read the previous section, it is immediately clear that a naïve compiler is not enough to get good performance. Implementing a Python compiler to get good performance is extremely challenging.

Since Python is dynamically typed, a static compiler can know little about the types of the variables in a program: hence, the compiler has little chances to discover short paths, thus inevitably incurring in the same problems described in Section 2.1.

Note that aggressive type inference does not help much, since unfortunately Python has not really been designed with this goal in mind, with the result that it is hard, if not impossible, to do it effectively: as described in section 2.1.5, the majority of entities in a Python program can change at runtime: calling function `fn()` could execute different code than the previous time, objects of class `cls` can suddenly get new attributes or methods that they did not have before, and so on.

The net result is that every time we reference a global entity, either directly (e.g. by calling a function) or indirectly (e.g. by invoking a method on an object whose class is known), little can be assumed about the result. In the past, there have been few attempts

to perform type inference for Python, but they did not work well [Can05] or they never saw the light [Sal04].

In conclusion, a static compiler would mainly solve the problem of the interpretation overhead described in section 2.1.1: hence, we can expect it to speed up programs by a factor of 2-2.5x, not more. Thus, to enhance performance we have to employ other techniques that go beyond a static compiler.

2.3 Our solution: automatically generated JIT compiler

As Section 1.2 explains, the two major strategies to implement dynamic languages are writing a fast interpreter, and writing a JIT compiler.

For the specific case of Python, writing a fast interpreter is not enough, because even if we manage to completely remove the interpretation overhead we can gain a limited speedup, as discussed in section 2.1.1.

Moreover, The static JIT compilation techniques employed by Jython and IronPython seems not to give good performance, as they rarely outperform CPython. Instead, what we need is a *dynamic JIT compiler* that behaves better than Psyco and solves its major drawbacks.

In particular, we seek a compiler that can be easily maintained and extended when the language is modified; moreover, it should be easy to port such a compiler to different target platforms, e.g. x86 and CLI, and to implement it for other dynamic languages.

The solution adopted by *PyPy* is to **generate automatically** a JIT compiler instead of writing it by hand. The language is implemented through a simple interpreter, which is then fed into a translator which augments it with many features, including a dynamic JIT compiler. The JIT generator is divided into a frontend and several backends, one for each target platform. The following chapters describes in detail the solution implemented by PyPy.

Chapter 3

Enter PyPy

3.1 What is PyPy?

The *PyPy* project¹ [RP06] was initially conceived to develop an implementation of Python which could be easily portable and extensible without renouncing efficiency. To achieve these aims, the PyPy implementation is based on a highly modular design which allows high-level aspects to be separated from lower-level implementation details. The abstract semantics of Python is defined by an interpreter written in a high-level language, called RPython [AACM07], which is in fact a subset of Python where some dynamic features have been sacrificed to allow an efficient translation of the interpreter to low-level code.

Compilation of the interpreter is implemented as a stepwise refinement by means of a translation toolchain which performs type analysis, code optimizations and several transformations aiming at incrementally providing implementation details such as memory management or the threading model. The different kinds of intermediate codes which are refined during the translation process are all represented by a collection of control flow graphs, at several levels of abstractions.

Finally, the low-level control flow-graphs produced by the toolchain can be translated to executable code for a specific platform by a corresponding backend. Currently, three fully developed backends are available to produce executable C/POSIX code, Java and CLI/.NET bytecode.

Although it has been specifically developed for Python, the PyPy infrastructure can in fact be used for implementing other languages. Indeed, there were successful experiments of using PyPy to implement several other languages such as Smalltalk [BKL⁺08], JavaScript, Scheme and Prolog [BLR09].

¹<http://codespeak.net/pypy/>

3.2 Architecture overview

PyPy is composed of two independent subsystems: the *standard interpreter* and the *translation toolchain*.

The **standard interpreter** is the subsystem implementing the Python language, starting from the parser ending to the bytecode interpreter. It is written in RPython, a language which can be easily be translated into lower-level and more efficient executables.

RPython is *statically typed*, and the types are deduced by type inference. The main restrictions compared to Python comes from the fact that the type system has been carefully designed in a way that the programs can be efficiently translated to C, JVM or CLI. For example, it is forbidden to mix integers with strings in any way, and similarly it is forbidden to change the layout of classes at runtime, such as adding or removing methods and attributes². RPython is a proper subset of Python, in the sense that if a Python program is RPython, it is guaranteed to have the same semantics both when translated and when interpreted by a standard Python interpreter.

Because of this, the standard interpreter can also be directly executed on top of any implementation of the language, e.g. CPython or PyPy itself: this is immensely useful during the development, because debugging a Python program is much easier than debugging the equivalent written in C. Obviously, the standard interpreter is very slow when executed this way, because of the huge overhead of the double interpretation.

Thus, we can think of the standard interpreter as an *high-level, executable and testable specification* of the language.

Moreover, since RPython is a high-level language, the standard interpreter is free of the many low-level aspects that are usually hard coded into the other Python implementations, such as e.g. garbage collection strategy or threading model.

These aspects are woven in at *translation time* by the **translation toolchain**, whose job is to turn the high-level specification into an efficient executable. Thus, the translation toolchain is much more than a compiler as we usual think of it: not only it translates the RPython source code to the target language, but it also implements the runtime system needed to execute a full-fledged virtual machine.

The target platforms are divided into two big categories: *lltype* and *ootype* (respectively for **low level** and **object oriented typesystem**). The choice of the type system determines the internal representation of the programs being translated. They mainly differ in their primitives: on the one hand the building blocks of *lltype* are structures and pointers, on the other hand *ootype* is about classes, methods and objects.

²Note that, although RPython is a limited subset of Python, it is only used internally as the implementation language of the standard interpreter, which in turn fully supports the whole Python semantics.

Finally, the translation toolchain transfers the control to one its *backends*, which are responsible to actually generate the final executable. Throughout the text, we will refer to `pppy-xxx` to indicate the executable produced by the backend *xxx*. At the moment of writing, there are three maintained backends:

- The **C backend** is based on *lltype* and emits C source code, which is in turn compiled by either `gcc` or *Visual C++*³. The produced executable is equivalent to CPython, as it is a native executable for the target platform, which currently can be *Linux*, *Mac OS X* or *Microsoft Windows*.
- The **CLI backend** [Cun06] [AACM07] is based on *ootype* and emits IL code for the CLI, i.e. the virtual machine, at the core of the *.NET Framework*. Currently, both *Mono* and *Microsoft CLR* are supported. The produced executable is roughly the counterpart of *IronPython*, although the latter is much better integrated with the hosting platform.
- The **JVM backend** [AACM07] is also based on *ootype* and emits bytecode for the JVM. Although the backend is complete, the resulting `pppy-jvm` is of little practical use because it still cannot access the hosting Java environment. It aims to be the equivalent of *Jython*, a Python implementation written in Java and fully integrated with the JVM.

3.3 JIT compiler generator

One of the most interesting components of the translation toolchain optionally generates a *JIT compiler* from the source code of the interpreter, in an automated way.

From the end user point of view, the presence of the JIT compiler is completely transparent⁴. The executable contains both the original interpreter and the generated JIT compiler: the code starts being interpreted, then the JIT automatically compiles the so called *hot spots*, i.e. the parts of the program that are executed more often and thus are most useful to optimize. The generated compiler is a *tracing JIT*: Chapter 5 describes the general idea behind it, and the current state of the art.

From the language implementor point of view, the generation of the JIT compiler is also mostly transparent: the programmer only needs to annotate the source code of the inter-

³The generated code is *ANSI C*, but makes optionally use of some specific compiler extension depending on the exact compiler we are using.

⁴Actually, there are hooks that the end user can call to tune the various parameters of the JIT, even though this it is not strictly necessary.

preter with few *hints* to drive the JIT compiler generator. These hints will be covered in details in Chapter 6.

The JIT compiler generator is divided into a *frontend* and several *backends*: to avoid confusion with the translation backends described above, we will refer to these as *JIT backends*.

The frontend contains all the architecture independent code: its job is to analyze the interpreted running program, find the hotspots, and optimize them. Its final result is a low level architecture independent representation of the program to compile: the actual generation of the executable machine code is done by the JIT backend. At the moment of writing, there are two maintained JIT backends:

- the **x86** JIT backend, which generates code for the *IA-32* architecture, i.e. for all the common 32 bit Intel-compatible processors around. In the future, there will probably be an *x86_64* JIT backend to exploit the new instruction set of the 64 bit processors
- the **CLI** JIT backend, which emits bytecode for the CLI virtual machine of the .NET Framework. The generated bytecode will in turn be translated into machine code by the .NET's own JIT compiler (see Section 4.2 for details).

It is obvious that the choice of the JIT backend is dependent on the choice of the translation backend: in particular, the x86 JIT backend is usable only in conjunction with the C translation backend, and the same for the CLI JIT backend and its homonym translation backend.

From an implementation point of view, the CLI JIT backend represent one of the major contributions of this thesis, and will be described in detail in Chapter 7.

3.4 Why choosing PyPy

In summary, we think that PyPy is a very good choice for the kind of research we are interested in. First of all, it is a mature project with an already working infrastructure and a vibrant (although not so large) development community, which is ideal to develop new ideas and solutions.

Moreover, the architecture and the modularity of the codebase allows us to concentrate on the main issues we are interested in: in particular, we do not have to worry about the complex semantics of Python, or to implement all the well known compilation techniques that form the starting point for the “interesting” work. In addition, by reusing PyPy we can apply the JIT compilation techniques described in this thesis to a real language, making the evaluation much more trustworthy than if we applied them to e.g. a toy language.

Last but not least, by working on a meta level the final result is more than just a fast implementation of Python: by providing a multiplatform JIT compiler generator, PyPy may become a very appealing framework to implement in a simple way all kinds of fast and portable dynamic languages: see for example [BKL⁺08] for a description of the implementation of Smalltalk and in PyPy, and [BV09] for a description of PyGirl, a Gambeboy emulator.

Chapter 4

Characterization of the target platform

From an implementation point of view, this thesis is about a dynamic JIT compiler for the .NET platform, and in particular for its *Virtual Machine* (VM), the CLI. However, our research is not strictly limited to the CLI, but it is potentially applicable to all the VMs which are similar enough.

What does “*similar enough*” mean? Defining the concept is hard, because even a small difference about a particular feature of the VM can have a big impact on the implementation strategy. On the other hand, some of the solutions proposed in this thesis can be applied equally well to virtual machines such as e.g. *LLVM*, whose underlying philosophy is very different from the CLI’s one, but still share some commonality. LLVM [LA04] stands for Low Level Virtual Machine and, as the name suggests, it is a virtual machine whose instruction sets is particularly close to the metal, contrarily to object oriented virtual machines whose instruction set is more high level.

This is especially true when speaking about performance: when targeting a VM, it is hard to accurately predict the performance behavior of the compiled programs, as they are going to be executed on a number of different implementations of the virtual machine. For example, .NET programs can be run either under the CLR, i.e. the original implementation from Microsoft, or under Mono, an alternative open source implementation. Moreover, all the implementations rapidly evolve over time, each new version providing slightly different performance of each feature or construct of the VM.

However, for languages implementors, this is far from being ideal: during the development of a compiler, they need to take a huge number of decisions about which constructs to use and which not to use to implement each particular feature, but their assumption might not be valid for alternative implementations or newer/older versions of the VM. As we will

see next in this chapter, this is especially true for more esoteric features or for constructs used in a slightly different way than they have been designed.

4.1 Dynamic loading of new bytecode

By definition, a JIT compiler emits new code during the execution of the program: thus, the first and most important requirement that a VM needs to fulfill in order to apply this research is the ability of generating and loading new bytecode at runtime, although the exact details vary between the CLI and the JVM.

For the CLI, the standard library of the .NET Framework, provides the necessary tools to generate and load single methods, in the namespace `System.Reflection.Emit`: in particular, by instantiating the class `DynamicMethod` it is possible to create new methods that are not bound to any particular class.

On the other hand, in the JVM the minimum unit of loading is the *class*: by writing a *custom classloader*, it is possible to generate and load new classes, and hence new methods, on the fly. Moreover, there are external libraries such as *ASM*[asm] and *BCEL*[bce] that simplify the task of generating and loading these classes.

4.2 JIT layering

Another important feature shared by the CLI and the JVM is the presence of a JIT compiler that translates the intermediate code into executable machine code¹.

For the cases we are analyzing, the JIT compiler generated by the PyPy translation toolchain emits code in the intermediate format proper of the hosting virtual machine. Then, this intermediate code is in turn compiled into executable machine code by the JIT compiler of the VM itself.

Thus, before being translated to executable machine code, the source code of our programs passes through two different JIT compilation *layers*:

- the **high level layer**, implemented by the PyPy translation toolchain
- the **low level layer**, implemented by the VM itself

¹The specifications of both the CLI and the JVM does not mandate the presence of a JIT compiler, which should be considered an implementation detail. However, all the current implementations of the CLI employ a JIT compiler, as well as all the most popular ones of the JVM.

JIT layering is a novel concept introduced with this thesis. In theory, if the low level JIT compiler were good enough, it could produce optimal code for whatever construct it encounters; in practice however, this rarely happens because either the low level JIT does not employ advanced techniques (as it is the case of the CLI), or it cannot have a deep understanding of the language semantics, thus missing a lot of optimization opportunities (as proved by the current implementations of dynamic languages for the JVM).

By adding an additional JIT compilation layer, specialized for a specific high-level language, much better and efficient code can be generated without modifying the underlying VM. This has several advantages:

- since the high level JIT compilers do not touch any of the internals of the VM, it is automatically portable across multiple implementations of the VM itself
- usually, the existing VMs and their corresponding JITs are very complex pieces of software, hard to modify: by writing our JIT on top of that, we avoid this problem; moreover, this way it is much easier to experiment with new features
- for the same reason, our approach is the only viable solution in cases we do not have access to the codebase of the VM, as in the case of Microsoft .NET

The main drawback of this approach is that the high level JIT compiler adds a overhead: if on the long run the time spent in the compiler is negligible compared to the time saved by running the optimized code instead of the non optimized one, in the short run programs could be slower.

As we will show in Chapter 8, this thesis proves that this approach is effective, and that the resulting implementation of the language can be much faster than a more ordinary implementation which relies only on the low level JIT. At the same time, we will also see that the overhead of the high level JIT compiler is not worth of in case of short running programs.

4.3 Immutable methods

As we seen in Section 4.1, both the CLI and the JVM offer the possibility of generating and loading new code at runtime. However once it has been loaded, it is not possible to modify the code inside methods.

In particular, both VMs do not offer any support for incremental lazy compilation. For example, there are cases in which we do not want to (or we cannot) eagerly generate the bytecode for all the possible code paths inside a method: the usual solution is to generate

the code only for the hot paths, and delay the generation of the others until they are reached for the first time, or until they have proved to be hot enough to justify the cost of the compilation.

Unfortunately, since it is not possible to modify a method, such a strategy cannot be easily implemented. As we will see in Section 7.5, this restriction is a serious limitation for language implementors who want to use adaptive techniques, and the proposed solutions (or, better, workarounds) either have a negative impact on the time spent for the high level JIT compilation or on the efficiency of the generated code.

4.4 Tail calls

On the CLI, we can explicitly mark a method invocation as a *tail call*, assuming that it is in *tail position*². Tail calls behaves exactly as normal calls, with the difference that the *call frame* of the caller is removed from the stack and replaced by the call frame of the callee: the result is that we can have an arbitrary deep chain of tail calls without any risk of exhausting the stack, as it would happen with normal calls. This process is called *tail call optimization*. Many functional programming languages such as Lisp, Scheme or Caml implement tail call optimization very efficiently, so that tail calls are as efficient as *gotos* [Ste77].

As we will see in Section 7.5, the problem of immutable methods could be partly solved by the presence of efficient tail calls. However, this is not the case for the current VMs:

- In the current implementations of the CLI tail calls are too slow to be used in practice. In particular, on Microsoft CLR tail calls are about 10 times slower than normal calls, while on Mono they are not even implemented correctly; in either case, they are orders of magnitude slower than a simple *goto*, making the `tail.call` instruction unusable for code that needs to be executed often.
- At the moment Java HotSpot does not support tail call optimization (see for example [Sch09] for a description of a possible implementation).

4.4.1 Tail calls benchmark

Figure 4.1 shows the source code used to benchmark the efficiency of tail calls. Both static methods compute the sum of the first n integers, the first using a loop, the second using tail recursion. Note that, although the code in the figure is written in C# for clarity, the

²A call is in tail position if it is immediately followed by a *return* instruction

```

public static int loop(int n)
{
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return sum;
}

public static int tailcall(int n, int sum)
{
    if (n < 0)
        return sum;

    // note: C# does not support the tail.call instruction
    return tailcall(n-1, sum+n);
}

```

Figure 4.1: *Tail call benchmark*

language does not support the `tail.call` instruction, so we had to manually write the algorithm directly in IL bytecode.

If tail call optimization is applied correctly, we would expect both versions to have about the same performance. However, this is not the case: on Mono, the recursive version is about 2.3 times slower than the loop, while on the CLR it is about 18.3 times slower.

Moreover, the Mono implementation of tail calls is known to be buggy[bug]: each tail call leaks a bit of stack space, with the result that a deeply nested calls might result in a stack overflow. As we will see in Section 8.5, this might trigger a crash in the code generated.

4.5 Primitive types vs reference types

Both the CLI and the JVM support values of primitive types and of reference types³:

- *primitive types* include numeric values, such as integers and floating point numbers of various sizes
- *reference types* include objects, that is, class instances

³Actually, the CLI also supports value types, enumerations, generic types, etc., but they outside the scope of this paragraph.

Although values of primitive types are not objects, it is possible to convert them through the mechanism of *boxing*: for each primitive type, there is a corresponding boxed type which wraps the value into a proper object. Once you have a boxed object, it is possible to get its original value by *unboxing* it.

Unfortunately, arithmetic operations between boxed values are much slower than between primitive values, due to the extra level of indirection and to the fact that it is necessary to allocate a new object on the heap to hold every intermediate result. Thus, to get high performance it is very important to use primitive types instead of boxed whenever it is possible.

4.6 Delegates (or: object oriented function pointers)

In the CLI, a *delegate* is a special kind of object that wraps a method (either instance or static): once created, the delegates can be freely stored and passed around, just as normal objects. The only method exposed by delegates is `Invoke`, which calls the wrapped method.

Delegates are type-safe: to instantiate one, we first need to create a *delegate type*, i.e. a class which inherits from `System.MulticastDelegate` and specifies its exact signature. Then, the VM can check that the signature of the delegate matches the signature of the method being wrapped.

Moreover, it is possible to bind a delegate to an object, which will be automatically passed as the first argument to the method when the delegate is called. This is a limited form of *closure*, which is usually exploited to create delegates that invokes an instance method on a specific object.

The JVM does not offer anything similar to delegates natively. However, it is possible to implement them by using the classical *Command* design pattern [GHJV93], i.e. by defining an interface for each signature we are interested in, and a class that implements it for each method we want to invoke.

Chapter 5

Tracing JITs in a nutshell

5.1 What is a tracing JIT compiler?

There are two general well known rules to consider when tackling the problem of compiler optimization:

- The **Pareto principle**[Wik10b], or *80-20 rule*: the 20% of the program will account for the 80% of the runtime. In other words, when executing a program most of the time will be spent in few regions, called *hot-spots*.
- The **Fast Path principle**[HH93]: it is usually possible to split expensive operations into a *fast path* and a *slow path*: the former handles the most common cases and, therefore, is expected to be executed for most time and has to be highly optimized, whereas the latter deals with the remaining cases and is not required to be particularly efficient.

Tracing JIT compilers are designed to exploit these rules to produce better and more efficient code. In particular, they are built on the following basic assumptions:

- programs spend most of their runtime in loops
- several iterations of the same loop are likely to take similar code paths

The basic approach of a tracing JIT is to generate machine code only for the hot code paths of most frequently executed loops and to interpret the rest of the program. The code for those common loops however is highly optimized, including aggressive inlining of the fast paths.

Tracing optimizations were initially explored by the Dynamo project [BDB00] to dynamically optimize machine code at runtime. Its techniques were then successfully exploited to implement a JIT compiler for a Java VM [GPF06, GF06]. Subsequently these tracing JITs were discovered to be a relatively simple way to implement JIT compilers for dynamic languages [CBY⁺07]. The technique is now being used by both Mozilla's TraceMonkey JavaScript VM [GES⁺09] and has been tried for Adobe's Tamarin ActionScript VM [CSR⁺09].

This chapter gives an overview of how a typical tracing JIT compiler works: it is not meant to be an exhaustive survey on *all* the various strategy adopted by known implementations, but it is useful to introduce some key concepts that are needed to understand the next chapter, which explains how the PyPy JIT compiler works.

5.2 Phases of execution

Typically, a tracing virtual machine goes through various phases when executing a program:

Interpretation When the program starts, everything is interpreted. The interpreter behaves like a standard one, except for the addition of some lightweight code to profile the execution, and in particular to establish which loops are run most frequently. To detect loops, it associates a counter to each backward jump, which is incremented every time the jump is taken. When the counter hits a certain threshold, the VM enters the *tracing* phase.

Tracing During this phase, the interpreter records a history of all the operations it executes. It traces until it has recorded the execution of one iteration of the hot loop¹. To decide when this is the case, the trace is repeatedly checked as to whether the interpreter is at a position in the program where it had been earlier. The history recorded by the tracer is called a *trace*: it is a list of operations, together with their actual operands and results. The trace is passed to the JIT compiler.

Compilation The job of the JIT compiler is to turn a trace into efficient machine code. The generated machine code is immediately executable, and can be used in the next iteration of the loop.

Running In this phase, the code generated by the JIT compiler is executed. Remind that a trace represent a *loop*, thus the code runs repeatedly until some exit condition

¹In case the trace becomes too long and its length exceeds a certain threshold, tracing is aborted and the control returns to the interpreter. This can occur e.g. in the unlucky case in which the iteration of the loop that is being traced is the last, and thus the backward jump is never taken.

is triggered. When the loop exits, the VM either goes back into the *interpretation* mode or transfers the control to another already compiled piece of code.

Being sequential, the trace represents only one of the many possible paths through the code. To ensure correctness, the trace contains a *guard* at every possible point where the path could have followed another branch, for example at conditions and indirect or virtual calls. When generating the machine code, every guard is turned into a quick check to guarantee that the path we are executing is still valid. If a guard fails, we immediately quit the machine code and continue the execution by falling back to interpretation.

Depending on the tracing technique, every guard might have an associated *guard failure counter*, similar to the loop counter seen above. The failure counter is incremented every time the guard fails, until it hits a certain threshold and the guard becomes *hot*. Then, we assume that the particular code path which starts from the guard failure is executed often enough to be worth compiling, and start tracing again from there. See Section 5.6 for more details.

Figure 5.1 summarize the transitions between the various phases of a tracing VM.

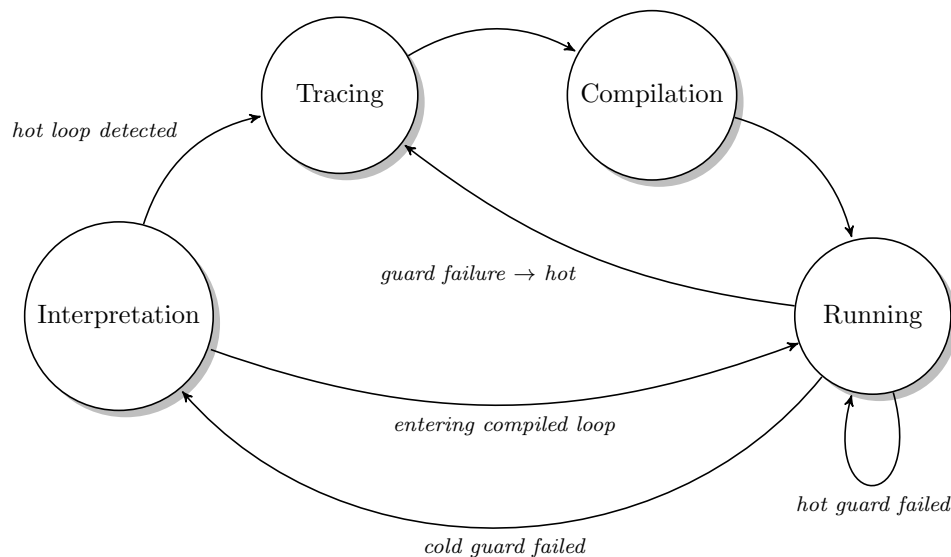


Figure 5.1: Various phases of a tracing VM

5.3 A deeper look at traces

In the previous section we have seen that a trace contains a history of all the operations executed, plus some *exit conditions* that are called *guards*. But, which kind of operations

are traced? And what is a guard, exactly? Giving a precise answer is difficult, as it depends on the details of the implementation: however, we can abstract some properties that are always valid.

To understand which kind of operations are traced, we need to recall that **tracing** is performed by an interpreter: every time the interpreter executes the next instruction, it also add an entry to the trace. Thus, we can think of it as an unrolled version of the original code, where the operations recorded in the history correspond to the opcodes encoded in the bytecode.

Moreover, for each operation we also record its operands, as well as its result. For each of them, we record both the origin (e.g. if it is the result of a previous operation) and its actual value (e.g., 42). Thus, a typical operation in the trace could look like `c[42] = add(a[30], b[12])`, where *a*, *b* and *c* are variables, whose value at the moment of execution is indicated inside the square brackets.

Traces are meant to be *linear*, i.e. a sequential list of all the operations that were actually executed, no matter where they come from. Thus, unconditional jumps are not recorded: instead, we simply continue the execution and record the subsequent operations. The same principle applies to *direct calls*, i.e. all calls in which the target function is statically known: the tracer does not record the call itself, but the operations executed inside the called function: this way, we automatically get the effect of *inlining*.

As long as we only encounter unconditional jumps, there is only one possible code path to be taken, so we are sure that all the subsequent iterations of the loop will follow the same one. Obviously, this is no longer valid as soon as we execute a *conditional* jump, i.e. a jump whose target destination depends on the runtime value of some condition. To ensure a correct behaviour each time a conditional jump is encountered, a *guard* is checked to ensure that the condition is still valid, and thus that the code path which is being executed is the correct one.

The same technique is used to follow calls whose target is not statically known. In particular, for calls through a function pointer, which we call *indirect calls*, the actual *value* of the pointer is guarded, while for *virtual method calls* the guard is on the exact *class* of the receiver object.

5.4 Example of tracing

In this section, we examine a concrete example of tracing to see how it works and how the resulting trace might look like.

Figures 5.2 and 5.3 show the source code and the corresponding bytecode for a simple

```

interface Operation {
    int DoSomething(int x);
}

class IncrOrDecr implements Operation {
    public int DoSomething(int x) {
        if (x < 0)
            return x-1;
        else
            return x+1;
    }
}

class tracing {
    public static void main(String argv[]) {
        int N = 100;
        int i = 0;
        Operation op = new IncrOrDecr();
        while (i < N) {
            i = op.DoSomething(i);
        }
        System.out.println(i);
    }
}

```

Figure 5.2: *Source code of the tracing example*

programs that contains a frequently executed loop, a virtual method call and a conditional branching instruction. We chose to use Java for these examples because it is a widely known programming language and because a lot of research on tracing JIT compilers has been done for the JVM. Moreover, by using Java we emphasize that this chapter describes the general techniques about tracing JIT compilers and not their implementation in PyPy, which is the subject of Chapter 6.

For the purposes of this example, we set the *hot loop threshold* to 3. Thus, the **tracing** phase begins at the fourth iteration of the loop, at line 19 in the Java code and 46 in the bytecode.

Figure 5.4 shows the recorded trace: to ease the reading, it shows the individual JVM instructions which are actually recorded side by side with the corresponding Java source code. We can see that most instructions are recorded as they are executed: however, few of them are not recorded at all, or have been replaced with something different. In particular:

- *Conditional jumps* are replaced by *guards* that check the expected condition is true. It is interesting to note that conditional jumps that are expected **not** to be taken are

```
class IncrOrDecr {
    ...
    public DoSomething(I) I
        ILOAD 1
        IFGE LABEL_0
        ILOAD 1
        ICONST_1
        ISUB
        IRETURN
    LABEL_0
        ILOAD 1
        ICONST_1
        IADD
        IRETURN
}

class tracing {
    ...
    public static main([Ljava/lang/String;)V
        ...
    LABEL_0
        ILOAD 2
        ILOAD 1
        IF_ICMPGE LABEL_1
        ALOAD 3
        ILOAD 2
        INVOKEINTERFACE Operation.DoSomething (I) I
        ISTORE 2
        GOTO LABEL_0
    LABEL_1
        ...
}

```

Figure 5.3: *Bytecode of the tracing example*

Method	Java code	Trace	Value
Main	while (i < N) {	ILOAD 2	3
		ILOAD 1	100
		IF_ICMPGE LABEL_1	false
		GUARD_ICMPLT	
	i = op.DoSomething(i);	ALOAD 3	IncrOrDecr obj
		ILOAD 2	3
		INVOKEINTERFACE ...	
		GUARD_CLASS (IncrOrDecr)	
DoSomething	if (x < 0)	ILOAD 1	3
		IFGE LABEL_0	true
		GUARD_GE	
	return x+1;	ILOAD 1	3
		ICONST 1	1
		IADD	4
		IRETURN	
Main	i = op.DoSomething(i);	ISTORE 2	4
	}	GOTO LABEL_0	

~~INSTR~~: Instruction executed but not recorded

INSTR: Instruction added to the trace but not executed

Figure 5.4: Trace produced by the code in Figure 5.2

replaced by a guard that checks the *opposite* condition: in the example, `IF_ICMPGE` (“jump if great or equal”) is replaced by `GUARD_ICMPLT` (“check less than”)

- *unconditional jumps* (`IRETURN` and `GOTO` in the example) are just removed from the trace
- *virtual method calls* are replaced by a guard that checks the exact class of the receiver, and the content of the method is just appended to the trace. In the example, the `INVOKEINTERFACE` to `DoSomething` is turned into a `GUARD_CLASS(IncrOrDecr)`, then the body of `IncrOrDecr.DoSomething` is recorded into the trace

5.5 Generalization of the trace and code generation

Once the trace has been completed, we can finally compile it. The goal of the **compilation** phase is to generate efficient machine code that can be used to execute the next iterations of the loop that we have just traced. Remember that a trace represents a single, concrete iteration of the loop, thus before actually emitting the machine code, we *generalize* the trace in order to make it representing a wider range of the next iterations.

As usually happens in most situations, there is a tension between abstraction and efficiency:

- if we generalize too much, we might end up with sub-optimal machine code
- if we don't generalize enough, we might end up with code bloats, as the code generated is too specialized and cannot be reused for some or most of the subsequent iterations of the loop

Sometimes, finding the right generalization is easy. Consider for example the trace above in Figure 5.4: most of the operands of the instructions are concrete integers (e.g., 1, 3, 4), thus we can generalize them to variables of type `int` and be sure that the generated code is optimal for all the next iterations. However, sometimes the job is more complex, especially if we have complex inheritance trees: for example, if some concrete value seen in the trace is an instance of, say, class `A`, we might want to generate code that accepts only `A` instances, or we might want to generate code that is more reusable and accepts also instances of some superclass of `A`.

The concrete strategy to solve this problem vary between each implementation of a tracing JIT. In Section 6.5.1 we will see how PyPy deals with it.

5.6 Linear traces vs. trace trees

So far, we have only seen examples of linear traces, but there are situations where the generated code is not sufficiently efficient. For the next example, we consider the loop in Figure 5.5 and its bytecode in Figure 5.6, and assume that the *hot loop threshold* is again set to 3.

The **tracing** phase begins when `i == 3`, and produces the trace shown in Figure 5.7. As we can see, the concrete iteration of the loop follows the path through the **else** clause of the **if**, guarded by the `GUARD_NE` condition.

However, it is clear that while **running** the guard will succeed only for the odd iterations of the loop: when it fails, the VM switches back to the **interpretation** phase (following the *cold guard failed* arrow in Figure 5.1), until the end of the loop. Then, when it is time to reenter the loop, it reuses the already compiled machine code, switching again to the **running** phase (*entering compiled loop*)

Such a behaviour is far less than optimal, because we need to interpret half of the iterations through the loop, and moreover the switch between the two phases produces some overhead. This happens because one of the original assumptions, the *Fast Path Principle* (see Section 5.1) has been violated: in this case, there is not just *one* fast path, but two that are equally important.

```

public static void trace_trees() {
    int a = 0;
    int i = 0;
    int N = 100;

    while(i < N) {
        if (i%2 == 0)
            a++;
        else
            a*=2;
        i++;
    }
}

```

Figure 5.5: Tracing tree example

```

public static void trace_trees() {
    ...
    LABEL_0
        ILOAD 1
        ILOAD 2
        IF_ICMPGE LABEL_1
        ILOAD 1
        ICONST_2
        IREM
        IFNE LABEL_2
        IINC 0 1
        GOTO LABEL_3
    LABEL_2
        ILOAD 0
        ICONST_2
        IMUL
        ISTORE 0
    LABEL_3
        IINC 1 1
        GOTO LABEL_0
    LABEL_1
        RETURN
}

```

Figure 5.6: Bytecode for `trace_trees`

Method	Java code	Trace	Value
<i>trace_trees</i>	while (i < N) {	ILOAD 1	3
		ILOAD 2	100
		IF_ICMPGE LABEL_1	<i>false</i>
		<i>GUARD_ICMPLT</i>	
	if (i%2 == 0)	ILOAD 1	3
	ICONST_2	2	
	IREM	1	
	IFNE LABEL_2	<i>true</i>	
	<i>GUARD_NE</i>		
	a*=2;	ILOAD 0	3
		ICONST_2	2
		IMUL	6
		ISTORE 0	6
	i++;	IINC 1 1	4
	}	GOTO LABEL_0	

Figure 5.7: Trace produced by the code in Figure 5.5

To deal with such cases in an efficient way, we need to modify the data structure that contains the traces, in order to support not only linear traces, but also *trees* of traces: if we observe that a guard fails often enough, it becomes *hot* and we start tracing again from there (the arrow *guard failure* \rightarrow *hot* in Figure 5.1): then, we compile the trace and teach the existing machine code to call the new one when that particular guard fails in the future (i.e., to follow the *hot guard failed* arrow).

This way, after an initial phase of warm-up, all the hot paths ends up being compiled, and thus the loop can be executed entirely in machine code. Moreover, this technique allows the VM to automatically adapt itself to the behaviour of the program, in case it varies during the execution, because if a cold path starts to be executed very often, it will be soon or later turned into a hot path and optimized as well.

However, implementing this technique in a virtual machine like the CLI and the JVM is not easy at all, as it does not offer direct support to modify the behaviour of existing code. As we will see in Section 7.5, this problem that we had to face during the development of the CLI backend is one of the most challenging.

Chapter 6

The PyPy JIT compiler generator

6.1 There is a time for everything

Usually, the lifetime of a program is split into *compile time*, when the source code is translated into executable machine code, and *runtime*, when the machined code is executed. However, this is not the case with PyPy when the JIT is enabled, as the lifetime is split into *four* different steps:

Translation time The step in which the translation toolchain (see Section 3.2) generates the actual executable from the RPython source of our interpreter. In this phase, the *JIT compiler generator* runs and generates the tracing JIT compiler for our language.

Interpretation time The step in which the user program¹ is run by the interpreter. It comprises both the *Interpretation* and the *Tracing* bubbles in Figure 5.1.

Compile time The step in which the generated JIT compiler actually runs. It corresponds to the *Compilation* bubble in Figure 5.1.

Runtime The step in which the code generated by the JIT compiler is executed. It corresponds to the *Running* bubble in Figure 5.1.

From this description, it is clear that the only static step is the *translation time*, which is performed only once whenever the source code of the interpreter is modified. The remaining three steps are all dynamic, i.e. they are active when the user program is running.

¹In our terminology, the final user is the programmer which write programs in the language.

6.2 Architecture of the JIT compiler generator

Figure 6.1 shows a more detailed view of the architecture. At the top of the picture there is the *interpreter*, which is written in RPython and follows two independent paths through the translation toolchain: the *regular translation* step is always done, and produces an efficient version of the interpreter which is included in the final executable. However the path which is more interesting for our purposes goes through the *codewriter*, and is active only when the JIT is enabled.

The *codewriter* translates the source code of the interpreter into intermediate code that will be exploited to trace the execution of the user programs. The result is a set of *jitcodes*, one for each RPython function or method that can be traced² from the main interpreter loop: these *jitcodes* are compiled into the final executable as static data, i.e. prebuilt instances that are already fully initialized when the program starts. We can think of the *jitcodes* as something that expresses the interpreter in the instruction set of a custom virtual machine, used solely for the JIT's internal use.

It is important to underline that there are three different kinds of intermediate code, which are executed at different levels. In the case of the Python interpreter, there is the Python bytecode, which represents the user program and is executed by the interpreter which is compiled to CLI bytecode. The *jitcodes*, which represents the interpreter itself in a format that is understandable by the JIT frontend to produce CLI bytecode for the hot loops. The CLI bytecode, which is executed by the underlying virtual machine.

The tracing phase (see Section 5.2) is implemented by the *JIT frontend*, which can be thought of as the custom virtual machine which can execute the *jitcodes*. To draw a parallel with the examples seen in Chapter 5, in PyPy RPython plays the role of Java, the *jitcodes* of the JVM bytecodes, and the JIT frontend is the equivalent of the tracing part of the JVM.

During the tracing phase, the frontend records a history of all the operations it executes, i.e. a *trace* (see Section 5.3). Once completed, the trace is then passed to the selected *JIT backend*, which translates it into efficient machine code, which is finally executed at *runtime*.

Like the interpreter, the JIT frontend and backends are written in RPython, and are compiled to CLI bytecode. Thus, the final executable is composed of three parts: the interpreter, the JIT frontend and backend, and the static data, which includes the *jitcodes*.

²Only a subset of all the RPython functions of the interpreter is seen by the JIT, and thus *traceable*: when the tracer encounters a call to a non-traceable function, it inserts a `call` operation in the trace, instead of inlining the body.

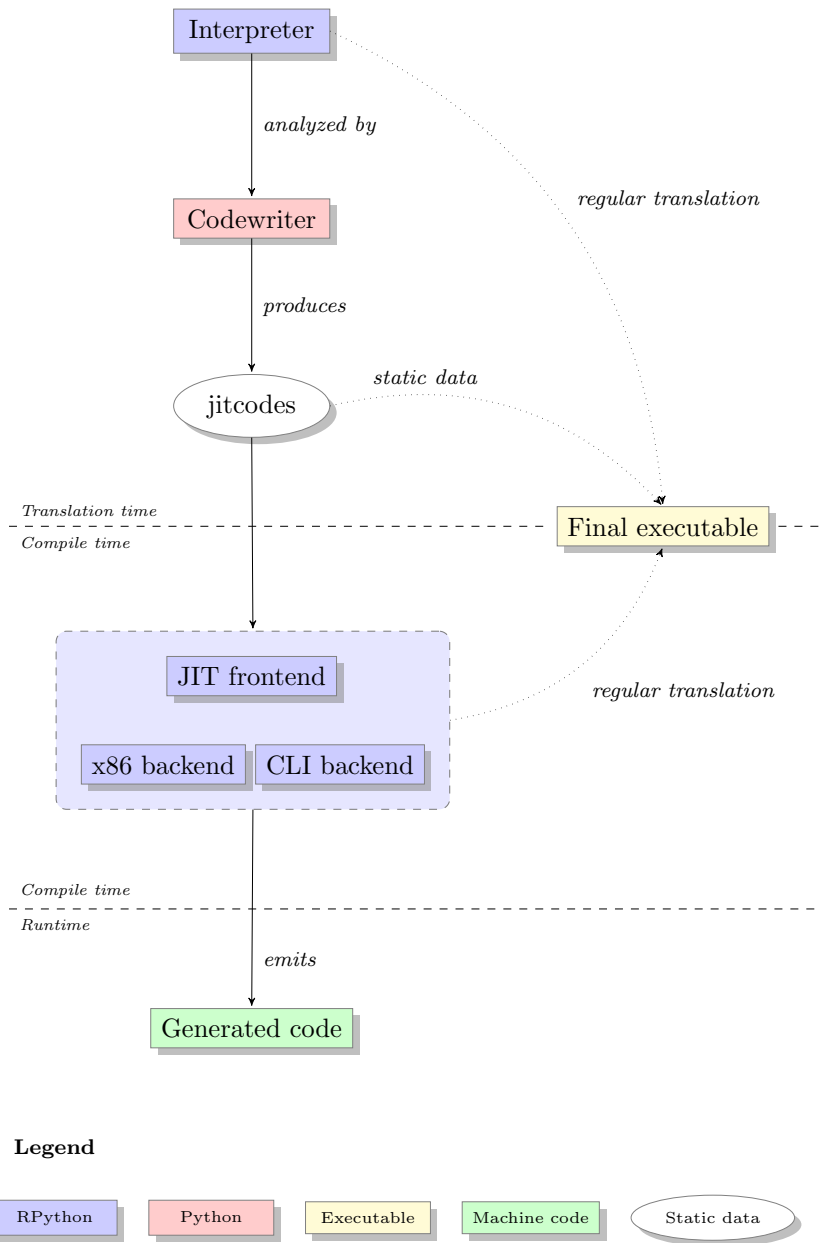


Figure 6.1: The architecture of the PyPy JIT compiler generator

6.3 Tracing the meta level

From the explanation above, it is clear that there is an important difference between the PyPy approach and the usual approach used by the other tracing JIT compilers described in the previous chapter.

Consider again the examples in Sections 5.4 and 5.6: during the interpretation phase the VM executes the bytecode instructions, which are also executed and recorded while tracing. On the other hand, in PyPy we do not trace directly the user program, but we *trace the interpreter* while executing the user program: in other words, the tracer operates at the meta level. This has already been described by the author of the thesis, together with other components of the PyPy team, in [BCFR09].

Tracing the meta level is the central idea that enables the generation of a JIT compiler for *all* the interpreters written in RPython, because the meta level is precisely what they have in common. As a result, the code is more modular and reusable, because the meta JIT compiler is strictly separated from the language and even unaware of its semantics. Moreover, the code is more portable to new target platforms and architectures, as proved by this thesis and the porting of the JIT to object oriented platforms and to the CLI in particular.

To avoid confusions between the two levels, we introduce some terminology to distinguish them. The *tracing interpreter* is used by the JIT to perform tracing. The *language interpreter* executes what we call the user's programs. In the following, we will assume that the language interpreter is bytecode-based. Finally, the *interpreter loops* are those contained in the language interpreter, while *user loops* are those in the user program.

6.3.1 Applying a tracing JIT to an interpreter

Unfortunately, we cannot directly apply the techniques described in Chapter 5 for tracing the simple interpreter defined in Figure 6.2: while the Pareto principle mentioned in Section 5.1 holds, because the language interpreter spends most of its execution time in the main loop switching to the current instruction to be interpreted (see Section 2.1.1), the Fast Path principle does not hold.

Indeed, each iteration of the main loop of the language interpreter corresponds to the execution of a single instruction, and it is very unlikely that the interpreter will keep executing the same instruction many times in a row.

An example is given in Figure 6.2. It shows the code of a very simple bytecode interpreter written in RPython with 256 registers and an accumulator. The `bytecode` argument is a string of bytes, all register and the accumulator contains integers. A program for this

```

def interpret(bytecode, a):
    """
    bytecode: string
        each instruction is composed by one char that indicates
        the opcode, and one char for the operand

    a: int
        initial value of the accumulator
    """
    regs = [0] * 256 # create a list of 256 ints initialized to 0
    pc = 0 # program counter
    while True:
        opcode = ord(bytecode[pc]) # cast the char into an integer
        pc += 1
        ## opcode dispatch: the chain of if...elif is turned into
        ## a switch statement by one of PyPy's optimizations
        if opcode == JUMP_IF_A:
            ## jump if the accumulator != 0
            target = ord(bytecode[pc])
            pc += 1
            if a:
                pc = target
        elif opcode == MOV_A_R:
            ## copy the accumulator into the specified register
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = a
        elif opcode == MOV_R_A:
            ## copy the specified register into the accumulator
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == ADD_R_TO_A:
            ## add the specified register to the accumulator
            n = ord(bytecode[pc])
            pc += 1
            a += regs[n]
        elif opcode == DECR_A:
            ## decrement the accumulator
            a -= 1
        elif opcode == RETURN_A:
            ## halt the execution and return the current value of the
            ## accumulator
            return a

```

Figure 6.2: A very simple bytecode interpreter with registers and an accumulator.

interpreter that computes the square of the accumulator is shown in Figure 6.3. If the tracing interpreter traces the execution of the `DECR_A` opcode (whose integer value is 7, hence the `guard_value` on `opcode0`), the trace would look as in Figure 6.4. PyPy traces are expressed in *Static Single Information Form* (SSI) [Ana99], i.e. all variables are assigned only once and gets renamed at branch points. Because of the guard on `opcode0`, the code compiled from this trace will be useful only when executing a long series of `DECR_A` opcodes. For all the other operations the guard will fail, which will mean that performance is not improved at all.

<code>MOV_A_R</code>	0	# <code>i = a</code>
<code>MOV_A_R</code>	1	# <code>copy of 'a'</code>
# 4:		
<code>MOV_R_A</code>	0	# <code>i--</code>
<code>DECR_A</code>		
<code>MOV_A_R</code>	0	
<code>MOV_R_A</code>	2	# <code>res += a</code>
<code>ADD_R_TO_A</code>	1	
<code>MOV_A_R</code>	2	
<code>MOV_R_A</code>	0	# <code>if i!=0: goto 4</code>
<code>JUMP_IF_A</code>	4	
<code>MOV_R_A</code>	2	# <code>return res</code>
<code>RETURN_A</code>		

Figure 6.3: Example bytecode: Compute the square of the accumulator

<code>loop_start(a0, regs0, bytecode0, pc0)</code>
<code>opcode0 = strgetitem(bytecode0, pc0)</code>
<code>pc1 = int_add(pc0, Const(1))</code>
<code>guard_value(opcode0, Const(7))</code>
<code>a1 = int_sub(a0, Const(1))</code>
<code>jump(a1, regs0, bytecode0, pc1)</code>

Figure 6.4: Trace when executing the `DECR_A` opcode

6.3.2 Detecting user loops

The key idea to solve this problem is to trace the user loops instead of the interpreter ones. One iteration over a user loop includes many iterations through the main loop of

the language interpreter. Thus, to trace a user loop we can *unroll* the main loop of the language interpreter until the user loop is closed.

How to detect when a user loop closes? User loops occur when the language interpreter executes twice an instruction stored at the same location. Typically, such location is identified by one or several variables in the language interpreter, for example the bytecode string of the currently executed function of the user program and the position of the current bytecode within that. In the example above, such variables are `bytecode` and `pc`. The set of all values that represent a particular position in the user program is called *position key*.

The tracing JIT does not have any knowledge of how the language interpreter it is implemented, hence it does not know the position key: thus, the author of the language interpreter has to mark the relevant variables with a *hint*.

The tracing interpreter will then effectively add the values of these variables to the position key. This means that the loop will only be considered to be closed if the position key assumes the same values twice. Loops found in this way are, by definition, user loops.

For obvious efficiency reasons the tracing interpreter does not check the position key after every instruction, but only when it encounters a backward jump. Since the tracing JIT cannot easily identify the code fragments where the interpreter implements backward jumps, the author of the language interpreter has to indicate such fragments with the help of a hint.

A similar technique is used to detect *hot user loops*: a counter is associated to each backward branch of the user program and it is incremented every time the branch is actually taken. When the counter reaches a certain threshold, the loop is considered hot.

The condition for reusing existing machine code also needs to be adapted to this new situation. In a classical tracing JIT there is no or one piece of assembler code per loop of the jitted program, which in our case is the language interpreter. When applying the tracing JIT to the language interpreter as described so far, *all* pieces of assembler code correspond to the bytecode dispatch loop of the language interpreter. However, they correspond to different paths through the loop and different ways to unroll it. To ascertain which of them to use when trying to enter assembler code again, the program counter of the language interpreter needs to be checked. If it corresponds to the position key of one of the pieces of assembler code, then this assembler code can be executed. This check again only needs to be performed at the backward branches of the language interpreter.

6.3.3 Applying the hints

Let us look at the example interpreter of Figure 6.2 to see where hints would be inserted. Figure 6.5 shows the relevant parts of the interpreter which have been annotated with

hints.

Initially, the class `JitDriver` has to be instantiated by listing all the variables of the loop. The variables are classified into two groups: *green* variables and *red* variables. The green variables are those identifying the position key, in this case `pc` and `bytecode`. All other variables are red. Red variables needs to be explicitly listed for purely technical reasons.

```
tlrjitdriver = JitDriver(greens = ['pc', 'bytecode'],
                          reds   = ['a', 'regs'])

def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        tlrjitdriver.jit_merge_point()
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                # apply the hint iif the jump is actually *backward*
                if target < pc:
                    tlrjitdriver.can_enter_jit()
                    pc = target
        elif opcode == MOV_A_R:
            ... # rest unmodified
```

Figure 6.5: *Simple bytecode interpreter with hints annotations*

In addition to the classification of the variables, there are two methods of `JitDriver` that need to be called. The first one is `jit_merge_point` which needs to be put at the beginning of the body of the main loop. The other is `can_enter_jit`, which is called at the end of any instruction which may implement a backward jump. In the example, only the `JUMP_IF_A` instruction can be a backward jump. Here is where the language interpreter performs profiling to decide when to start tracing. It is also the place where the tracing JIT checks whether a loop is closed. That is when the green variables assume values already recorded for an earlier call to `can_enter_jit`.

In this tiny example a considerable number of lines have to be added for the hints; however, in a real interpreter the number of added lines would be negligible in comparison with the length of the program, as Section 8.1 shows for the case of the Python Interpreter.

When executing the `Square` function of Figure 6.3, the profiling will identify the loop in the square function to be hot, and start tracing. It traces the execution of the interpreter running the loop of the square function for one iteration, and the unroll the interpreter

```

1 loop_start (a0, regs0, bytecode0, pc0)
2 # MOV_R_A 0
3 opcode0 = strgetitem(bytecode0, pc0)
4 pc1 = int_add(pc0, 1)
5 guard_value(opcode0, 2)
6 n1 = strgetitem(bytecode0, pc1)
7 pc2 = int_add(pc1, 1)
8 a1 = list_getitem(regs0, n1)
9 # DECR_A
10 opcode1 = strgetitem(bytecode0, pc2)
11 pc3 = int_add(pc2, 1)
12 guard_value(opcode1, 7)
13 a2 = int_sub(a1, 1)
14 # MOV_A_R 0
15 opcode1 = strgetitem(bytecode0, pc3)
16 pc4 = int_add(pc3, 1)
17 guard_value(opcode1, 1)
18 n2 = strgetitem(bytecode0, pc4)
19 pc5 = int_add(pc4, 1)
20 list_setitem(regs0, n2, a2)
21 # MOV_R_A 2
22 ...
23 # ADD_R_TO_A 1
24 opcode3 = strgetitem(bytecode0, pc7)
25 pc8 = int_add(pc7, 1)
26 guard_value(opcode3, 5)
27 n4 = strgetitem(bytecode0, pc8)
28 pc9 = int_add(pc8, 1)
29 i0 = list_getitem(regs0, n4)
30 a4 = int_add(a3, i0)
31 # MOV_A_R 2
32 ...
33 # MOV_R_A 0
34 ...
35 # JUMP_IF_A 4
36 opcode6 = strgetitem(bytecode0, pc13)
37 pc14 = int_add(pc13, 1)
38 guard_value(opcode6, 3)
39 target0 = strgetitem(bytecode0, pc14)
40 pc15 = int_add(pc14, 1)
41 i1 = int_is_true(a5)
42 guard_true(i1)
43 jump(a5, regs0, bytecode0, target0)

```

Figure 6.6: Trace when executing the Square function of Figure 6.3, with the corresponding bytecodes as comments.

loop of the example interpreter eight times. The resulting trace can be seen in Figure 6.6. The trace is divided into 8 different fragments, one for each opcode that were traced: each fragment begins by fetching the current opcode from `bytecode` and incrementing the program counter `pc`, then a guard checks the value of the opcode against its numeric value (e.g., `MOV_R_A` corresponds to 2, as can be seen in line 5). Moreover, additional guards are inserted at each branching point: for example, the `guard_true` at line 42 corresponds to the `if a` in Figure 6.5.

6.4 Loops, bridges and guards

So far, we know that a trace is a sequence of jitcode instructions. However, traces are classified in four different kinds (see Figure 6.7) accordingly to the way they are connected together:

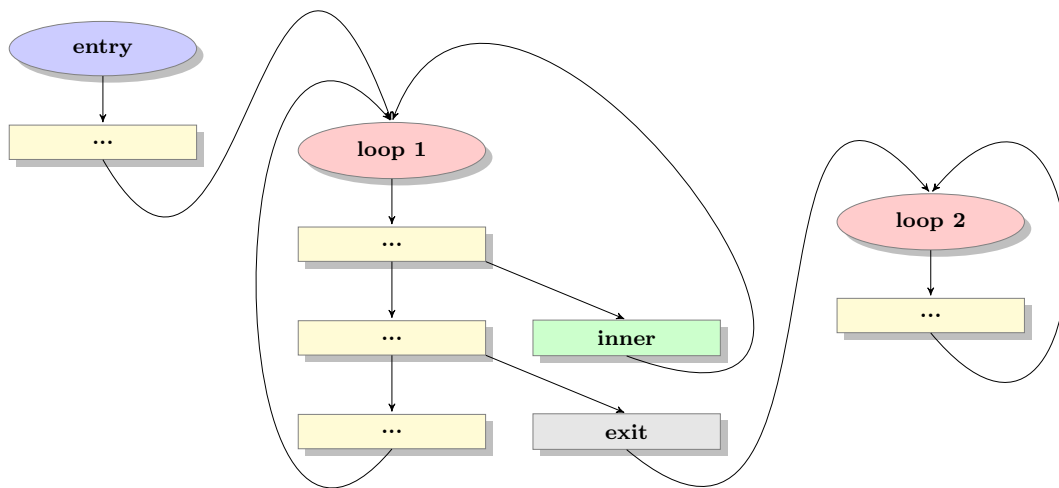


Figure 6.7: *Loops and inner, entry and exit bridges*

- **Loop**, drawn in red and yellow: a trace which implicitly jumps back to its top when it finishes. The only way to exit a loop is when a guard fails.
- **Entry Bridge**, drawn in blue and yellow: a trace which is executed only once, then jumps to a loop. Entry bridges are used to execute “setup code” before entering the real loop. See Section 6.5.2 for more details.
- **Inner Bridge** or simply **Bridge**, drawn in green: a trace that starts from a failing guard when it becomes hot. They are useful in case a loop has more than one fast

path. When entered, the code in a bridge is executed only once, then it jumps to the original loop.

- **Exit Bridge**, drawn in grey: like an inner bridge, but instead of jumping back to the original loop, it jumps to another one. Exit bridges share characteristics with both inner and entry bridges: like inner bridges they starts from a guard failure, and like entry bridges they jumps to a different loop at the end.

A loop is linear as long as all its guards are cold. When a guard becomes hot, an inner bridge is generated from there and attached to the guard itself: the idea is that when the guard fails, the execution is directly transferred to this new trace, without any need of exiting the *runtime* phase. Thus, we effectively get a tree of traces, as described in section 5.6.

As an example, take the code listed in Figure 6.8: it shows the very same algorithm as in Figure 5.5 written in RPython.

Since it is written in RPython it is an *interpreter loop*, following the definition given in Section 6.3: it can be thought as the equivalent of an interpreter main loop. For clarity, the hints `jit_merge_point` and `can_enter_jit` have been omitted. There are two different code paths in the loop, which are both frequently executed and, hence, hot.

```

a = 0
i = 0
N = 100
while i < N:
    if not i % 2:
        a += 1
    else:
        a *= 2
        i += 1

```

Figure 6.8: *RPython loop with two equally hot code paths*

Suppose that the *hot loop threshold* is set to 3. The tracing phase starts at the beginning of the fourth iteration, with i equal to 3: thus, the code will follow that path inside the *else* branch, producing the loop shown in Figure 6.9. There are several interesting details to note about that loop:

- two *guards* are produced: the first one corresponds to the condition of the **if**, the second one to the condition of the **while** loop. The conditions are in that order because tracing starts just **after** the loop has been entered, so the first statement it executes is the **if**
- the loop takes two parameters as input, called *input arguments*. Note that the *jump* at the end loops back passing the updated values for i and a

Note also that the loop contains the code to handle only the *else* branch of the **if**: when at runtime the condition is false, the *guard.true*(t_1) (highlighted in blue) fails³, and the execution is switched back to the interpretation phase. Eventually the guard becomes hot, so we start tracing from there and attach the newly created bridge to the existing loop,

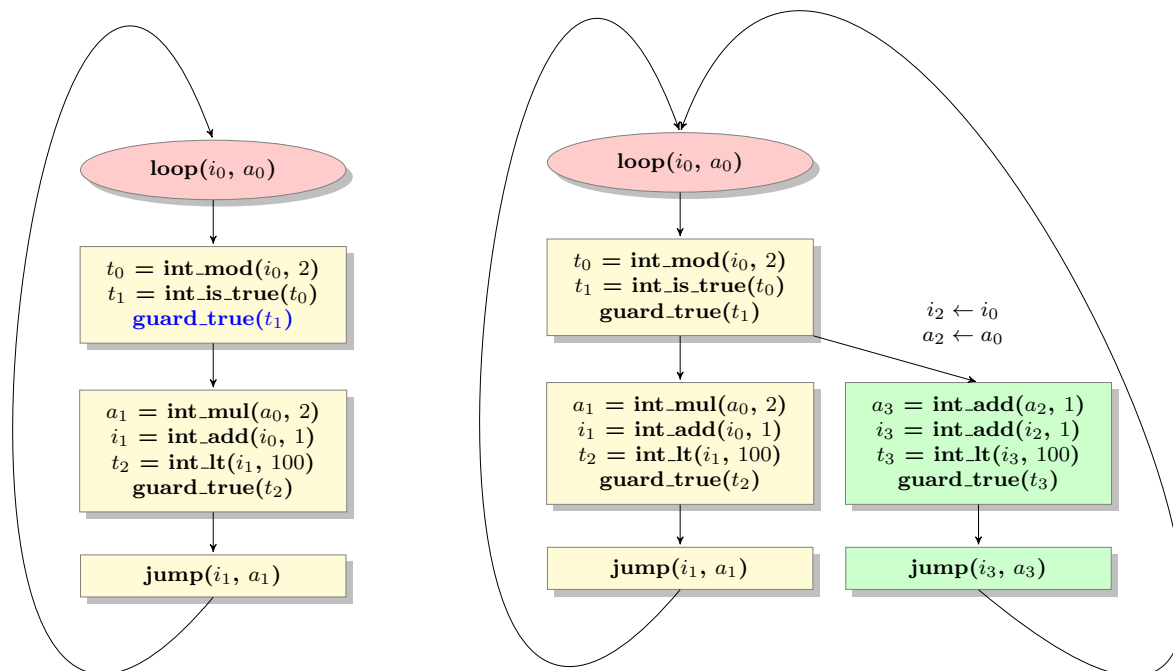


Figure 6.9: The trace produced by the code in Figure 6.8

Figure 6.10: The trace tree produced when the guard becomes hot

getting the result shown in Figure 6.10. The fact that one branch is in the main loop and the other is in the inner bridge is only due to the threshold being odd. If it were set to an even number, the two traces would be swapped.

It is important to note that the bridge needs to be compiled and attached to the main loop *after* it has already been compiled and executed for while: thus the machine code must be generated in a way that can be dynamically extended as needed. As we will see in Chapter 7, this requirement poses some problems for the CLI JIT backend.

6.5 Escape analysis and trace optimization

Once the trace has been created, it is passed to the *trace optimizer*, which does constant folding, removes unneeded operations, and more generally produces smaller and more efficient traces. For the most part, it is possible to exploit all the well known and standard techniques that have been developed during many years of research in compilers (see e.g.

³The original condition is `not i % 2` but the logical negation does not appear in the trace: this is due to an optimization that replaces pairs of `bool_not/guard_false` with `guard_true`.

[BGS94] for a survey on them).

6.5.1 Escape analysis

One of the most important optimizations is obtained by performing *escape analysis* [Bla99] [dCGS⁺99] to remove unnecessary allocations of objects. For example, consider the code in Figure 6.11, which shows a typical pattern that appears in dynamic languages. The `w_Int` class represents *boxed integers*: because of the arithmetic operation performed inside the loop, a new temporary `w_Int` instance is created at each iteration (see also Section 2.1.2).

The unoptimized loop produced during tracing is shown in Figure 6.12. The loop takes `obj0`, of type `w_Int`, as an input argument. In the first block, the class of `obj0` is guarded in order to safely inline the body of `getval` (as explained in Section 5.4), then the value for `nextval` is computed. Then, the second block allocates a new object of type `w_Int` and inlines the body of `__init__`, initializing the field `val`. Finally, the last block inlines again the body of `getval`, then it evaluates the condition of the loop.

A simple static analysis of the trace shows that the object allocated inside the loop **never escapes** outside. Thus, we can avoid to allocate it, and *explode* its fields into simple local variables. Thus, each `setfield_gc` to the object becomes an assignment to the corresponding variable, and accordingly for `getfield_gc`. Moreover, operations like `guard_class` can be removed as they are statically known to be true.

The resulting optimized loop is shown in Figure 6.13: note that instead of allocating a new object inside the loop, we reserve the space for all its field as local variables (in this case only one, `obj_val`, which gets renamed because of the *SSI* form). Accordingly, instead of having one input argument of type `w_Int`, we have as many input arguments as its fields: in this case, `obj_val0` is of type `int`.

Obviously, such an optimized loop works only if `obj` is already of type `w_Int` **before** entering the loop. Thus, we say that the loop is *specialized* for that type. If later in the execution of the program we try to enter again the loop but `obj` is of a different type than `w_Int`, we start tracing again and produce another specialized version of the loop.

```
class w_Int:
    def __init__(self, val):
        self.val = val

    def getval(self):
        return self.val

def f(n):
    obj = w_Int(n)
    while obj.getval() > 0:
        nextval = obj.getval() - 1
        obj = w_Int(nextval)
```

Figure 6.11: *Example of boxed arithmetic*

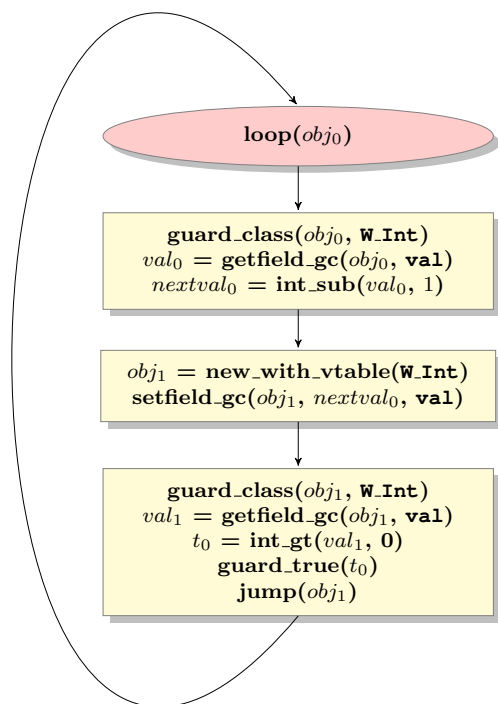


Figure 6.12: *Unoptimized trace: obj is allocated*

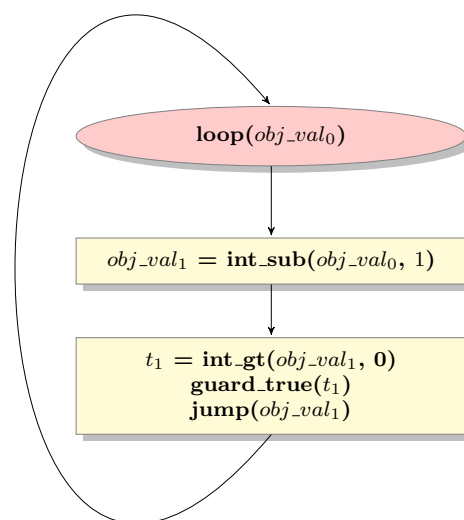


Figure 6.13: *Optimized trace: obj is virtualized*

The objects for which we avoid the allocation are called *virtual instances* and the same technique works also for arrays whose size is statically known at tracing time, which are called *virtual arrays*. Generically, we refer to either virtual instances or arrays with the term *virtuals*. To our knowledge, the JIT compilers generated by PyPy are the only tracing JITs that exploit escape analysis to optimize the generated code.

6.5.2 Reentering a specialized loop

Once the specialized loop has been compiled, we need a way to *enter* it. This is not straightforward, as now there is a mismatch between the layer of the interpreter, which gives us a real `W_Int` instance, and the layer of the loop, which needs an *unboxed* one.

Entering the loop for the first time it is easy: remind that during tracing we have actually *executed* one iteration of the loop shown in Figure 6.12, so we know the current value of `obj1.nextval`, which corresponds to `obj_val0` in Figure 6.13: therefore we can just enter the specialized loop passing it that value. The loop will run until one guard fails, then the execution of the program continues as usual.

However what if we want to re-enter the specialized loop a second time, later in the execution of the program? We still have the mismatch between the `w_Int` instance given by the interpreter and the virtualized one expected by the loop, but this time we do not know which value to pass it. One easy solution is to enter again the tracing phase in order to do one (unoptimized) iteration of the loop and compute the value for `obj1.nextval`, then discard the trace and enter the compiled loop.

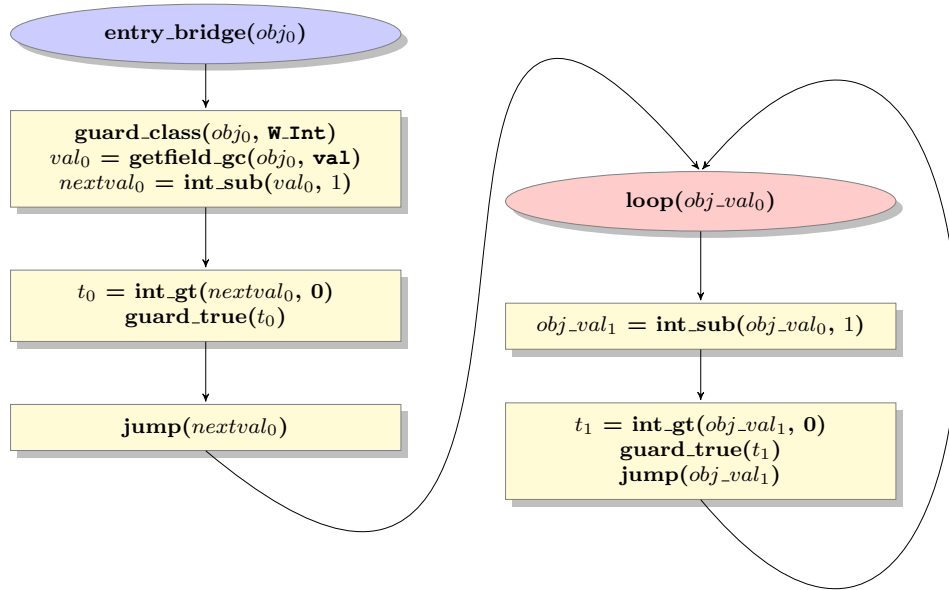


Figure 6.14: *The entry bridge that jumps to the optimized loop*

Clearly computing and discarding a new trace every time a loop is entered is not efficient. Therefore instead of being thrown away, it is compiled into an *entry bridge* (see Section 6.4) that then jumps directly to the specialized loop, as shown in Figure 6.14. Once it has been compiled, the entry bridge will be reused all the times it is necessary to re-enter the loop.

6.5.3 Forcing virtuals

Instances and arrays that are allocated inside the loop can be *virtualized* only if they don't *escape* the loop itself. But, what happens if later we add an inner bridge to the loop which contains some operations that make the object escaping?

Consider for example the code in Figure 6.15, where `external_func` is an arbitrary function that for some reason cannot be traced, and thus inlined, by the JIT. Calling external functions is one of the operations that make an object escaping.

If we call f with a large n , for the first iterations the condition of the **if** will never be true, thus the JIT sees a trace which is very similar to the one in Figure 6.12 and *virtualizes* `obj`.

Figure 6.16 shows the resulting loop: the yellow blocks represents the main loop, while the green blocks represents the bridge which is attached to *guard_false*(t_0), when it eventually starts to fail.

As long we stay in the main loop, `obj` is virtual. However, when we enter the bridge we need a *real* object of type `W_Int` to pass to `external_func`: thus, a new object is allocated and initialized using the values of its virtual fields, in this case *obj_val*₀. This process is called *forcing* of virtuals. Once the virtual has been forced, we can call the function and continue the execution as normal. This technique is very important to get good performance, as it allocates objects lazily only when it is really necessary.

```
def f(n):  
    obj = W_Int(n)  
    while obj.getval() > 0:  
        if obj.getval() < 10:  
            external_func(obj)  
            nextval = obj.getval() - 1  
            obj = W_Int(nextval)
```

Figure 6.15: *Example of virtual which is forced*

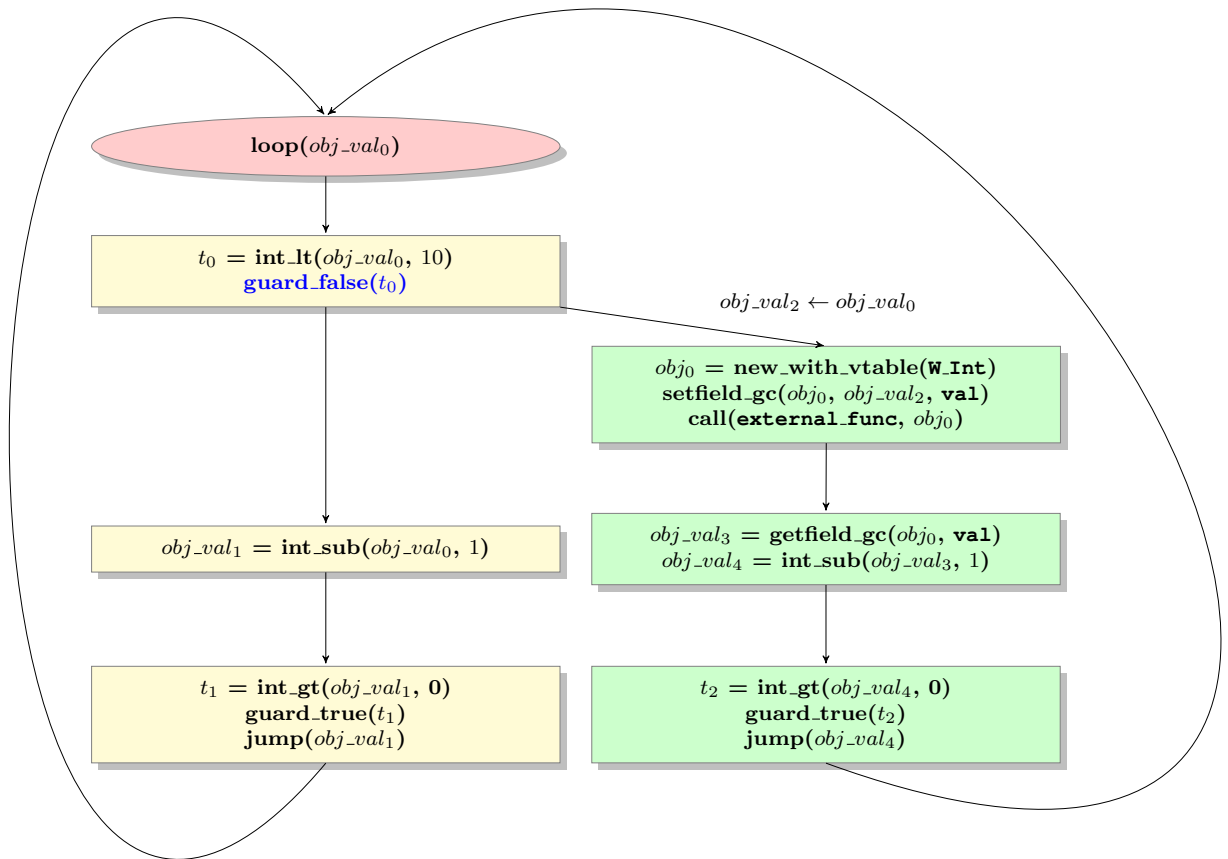


Figure 6.16: Loop that shows how virtuals are forced

Chapter 7

The CLI JIT backend

7.1 Teaching the JIT frontend Object Orientation

As explained by Section 3.2, the Translation Toolchain uses either *lltype* or *ootype* for its internal representation. Originally, the PyPy JIT compiler generator has been designed for *lltype*, in particular for being translated with the C backend and to emit machine code for a CPU like the *x86* instruction set. Before writing the backend for the CLI virtual machine, we had to port both the *codewriter* and the JIT frontend to *ootype*. In the spirit of Chapter 4, the porting has been done with both the CLI and the JVM in mind, with the idea that writing a JIT backend for the JVM should be very easy once the whole infrastructure is ready.

In particular, the most important difference between *lltype* and *ootype* is how to inline method calls. As we saw in Section 6.2, to inline a method call the tracer must know which *jitcode* it corresponds to. In general, to know which implementation of the method will be called on a specific object, we must know its class: thus, during the tracing phase, the tracer put a `guard_class` in front of each method invocation, which from that point on can be assumed to be known.

In *lltype* a method implementation is represented by a function pointer stored in the *vtable*¹ of the class, called using the `indirect_call` operation. Since the class, and hence its *vtable*, is known, we can easily discover the memory address of the target method. Then, we can look up a table, called `indirectcall_dict`, which maps memory addresses to *jitcodes* to find the desired one.

¹The *Virtual Method Table* or *vtable* is the standard mechanism to implement late binding in statically typed object oriented languages. The *vtable* is a per-class table of function pointers pointing to the correct implementation of each method for the given class.

```

class A:
    def f(self):
        return 1

    def g(self):
        return 2

class B(A):
    def f(self):
        return 3

```

Method <i>f</i>	
Class	<i>Jitcode</i>
A	A.f
B	B.f

Method <i>g</i>	
Class	<i>Jitcode</i>
A	A.g

Figure 7.1: RPython hierarchy of classes

Figure 7.2: Method tables for the jitcodes

In *ootype*, classes and methods are first class citizens: methods are invoked using the `oosend` operation, which takes care of dispatching to the correct implementation and is translated into a *virtual method call* on the target VM². Thus, the virtual method dispatching mechanism is hidden from us and neither the *vtable* nor the memory addresses of the method implementations can be accessed.

Therefore, instead of using the above mentioned `indirectcall_dict`, we build a different table for each possible method which maps the **exact class** of the object to the *jitcode* that corresponds to the right method implementation. Then, when the tracer needs to inline a method call, it can just look up the class of the object, or one of its superclasses, in the table for the method it is considering.

Figure 7.1 shows an example in RPython with one class `A` which defines two methods, and its subclass `B` which overrides one. Figure 7.2 shows the method tables computed for both methods. Being RPython, the set of methods for each class does not change at runtime (see Section 3.2), thus the method tables can be computed at translation time by the *codewriter*, while the frontend uses them at compile time.

7.2 The interface between the JIT frontend and backend

This section describes how the JIT frontend interacts with the backend: even if most of the job of the backend is done during the *compilation* phase (see Figure 5.1), we will see that it plays an important role also for the other phases of the tracing JIT. The frontend is designed to be as generic as possible, but there are tasks for which it is indispensable to

²`callvirt` on the CLI and `invokevirtual` on the JVM

interact directly with the target platform: thus, the backend connects the abstract layer used by the frontend and the concrete level of the target platform. In this section, we describe the interface for *ootype* backends. The interface for *lltype* backends is the same for the most part and the differences are minimal (for example, they do not have any notion of *oosend*).

Concretely, the backend must implement the `CPU` class: the frontend stores a reference to a singleton instance of this class, and invokes method on it. `CPU` is the only class seen by the frontend, but internally the code of the backend is divided into more classes. The subsequent subsections describes the interface of the `CPU` class. Then, the next sections of this chapter will explain how the CLI backend implements this interface.

7.2.1 Descriptors

As described in Section 7.1, the *ootype* version of the JIT frontend is aware of the object oriented entities of the target platform. However, the concrete internal representation of classes, methods, etc. used by *ootype* is obviously different from the representation used by the native tools of the platform. For example, in *ootype* the concept of “class” is denoted by instances of the class `ppypy.rpython.ootype.Instance`, while on the CLI side we use instances of `System.Type`.

The concept of *descriptors* fills the gaps between the two worlds: a descriptor is a token that can be obtained by the frontend at translation time. Such token is *opaque* and the frontend has no way to inspect its inside, and its only use is to be passed back to the backend later. The following table lists all possible descriptors:

Descriptor	Meaning
<code>typedescr(T)</code>	the type T
<code>arradescr(T)</code>	the array type whose items are of type T
<code>methdescr(T, name)</code>	the method name of the type T
<code>fielddescr(T, name)</code>	the field name of the type T
<code>calldescr(ARGS, RESULT)</code>	a static ³ method type of the specified signature

7.2.2 Executing operations

As already explained, during the tracing phase the operations are **both** executed and recorded. To guarantee correctness, the result of the execution must be the same that would be

³I.e., a method which is not bound to any particular instance. Static methods are the CLI/JVM equivalent of functions.

obtained by the underlying VM. For simple operations the JIT frontend knows everything it needs for execution: for example, the operation `int_add(40, 2)` yields always the same result whatever platform we are targeting.

However, some operations cannot be performed without knowing the implementation details of the target platform. Take, for example, the operation `getfield_gc`⁴, which reads the content of a field of an object: depending on the platform, it could be implemented in very different ways, e.g. by fetching a value stored in memory at a certain offset from the address of the object (as it is the case for the *x86* backend), or by invoking the reflection APIs of the runtime system to actually get the desired value (as the *CLI* backend might do, but the actual implementation is slightly different for efficiency reasons).

Therefore the JIT frontend has to delegate the execution of some operations to the backend. The needed extra information are passed as *descriptors*: following the `getfield_gc` example, the field to read from is encoded in a *fielddescr*.

The backend exports one method for each of the operations it implements, as shown by Figure 7.3. Each method takes a *descriptor* and a list of arguments, which contains the actual operands. The difference between `do_new_with_vtable` and `do_runtimew` is the same as between creating an object in C# using the statement `new` and creating an object using the reflection.

Note that these methods directly *execute* the operations, but do not emit machine code for them. Machine code generation is described in Section 7.3.

7.2.3 Loops and bridges

The most important task of the backend is to translate loop and bridges (see Section 6.4) into efficient machine code that can be directly executed on the target platform. To request the compilation of a loop, the frontend calls the method `compile_loop`, passing the trace, i.e. the list of operations to be emitted, and the list of *input arguments*. Despite the name for historical reasons, `compile_loop` is also called to compile entry bridges.

Similarly, to compile inner and exit bridges the frontend calls `compile_bridge`, which takes the same parameters as `compile_loop`, with the addition of a token that describes which *hot guard* the bridge starts from. The backend connects the existing guard and the newly created bridge so that the next time the guard fails, the bridge will be executed.

⁴the `_gc` suffix is there for historical reasons and makes sense only for *lltype*, thus can be ignored for *ootype*

Operation	Arguments	Descr	
do_new_with_vtable		typedescr	New object of type typedescr
do_new_array	length	typedescr	New array typedescr[length]
do_runtimew	class		New object of type class
do_getfield_gc	obj	fielddescr	Get the specified field out of obj
do_setfield_gc	obj, value	fielddescr	Set the specified field of obj to value
do_oosend	args_array	methdescr	Call the specified instance method. The receiver is in args_array[0]
do_instanceof	obj	typedescr	Test whether obj is an instance of typedescr
do_arraylen_gc	array	arraydescr	Return the length of array
do_getarrayitem_gc	array, i	arraydescr	Return array[i]
do_setarrayitem_gc	array, i, value	arraydescr	array[i] = value
do_call	args_array	calldescr	Call the static method in args_array[0]

Figure 7.3: *The operations implemented by the JIT backends*

7.2.4 Executing loops

Once compiled, a loop is ready to be executed: before doing so, the JIT frontend needs to pass it the correct *input arguments*, so that it starts in the right state. The backend exposes three methods for setting the input arguments:

- `set_future_value_int(i, value)`
- `set_future_value_float(i, value)`
- `set_future_value_ref(i, value)`

Each of them sets the *i*th input argument to *value*, and they only differ for the type of the value parameter⁵, which can be respectively `int`, `float` or `object`⁶.

Note that the type of *value* for `set_future_value_ref` is `object`: whenever an object is used as an input argument, it is casted to `object`, passed to `set_future_value_ref`, and casted back to its original type inside the loop. This has to be done because the frontend does not know in advance all the possible types that could be used for input arguments.

Once the input arguments have been set, the JIT frontend triggers the execution of the loop by invoking the `execute_token` method, which takes a single parameters that uniquely identifies the loop in question. Then, the loop runs until a guard fails.

⁵We need three distinct names because the C backend does not support the overloading of functions.

⁶The suffix `_ref` stands for “reference value”, i.e. an object for *ootype* and a pointer for *lltype*

At this point, the JIT frontend needs to know the values of the internal variables of the loop to restart the interpreter in the right state. These can be queried by invoking `get_latest_value_int`, `get_latest_value_float` and `get_latest_value_ref`, which work in a similar way as their `set_future_value_*` counterparts.

Note that in this way input arguments passing and value returning is not very efficient, as a method invocation is needed for each of them. However, tracing JITs spend most of their time **inside** the loops, so this process does not affect negatively efficiency.

7.3 Compiling loops

As anticipated by Section 4.1, the PyPy JIT emits new bytecode at runtime. In .NET, the unit of compilation is the *method*, hence each compiled loop generates a method.

The signature of *all* the generated methods is `void Loop(InputArgs args)`, expressed in C# syntax. The only parameter is of type `InputArgs`, which contains three arrays of integers, floats and objects. The CPU holds an instance of `InputArgs`, which is then passed to all the loops when invoked⁷. The methods `set_future_value_*` store the actual values of the input arguments in these arrays.

Despite its name, which survives for historical reasons, `InputArgs` is used **both** for input arguments and for return values. Thus, the generated code will store the relevant values into the same arrays, which will then be read by `get_latest_value_*`.

A generated method is composed of three parts:

- the **setup code**, which initializes all the local variables to their starting values, according to the content of `args`
- the **loop body**, which actually executes the loop. Generating the body is easy, as each operation can be easily mapped into a sequence of *CLI instructions*. Section 7.4 explains in detail how the code is actually generated
- the **tear-down code**: for each guard, there is a bit of code which is triggered when it fails and stores the relevant local variables into `args`

⁷Thus, since CPU is a singleton, so is `InputArgs`. This works well as long as we support only one thread, but in case of multi-threading we may want to have a per-thread `InputArgs` instance

```

Not = ['ldc.i4.0', 'ceq']

operations = {
    'same_as':      [PushAllArgs, StoreResult],
    'bool_not':    [PushAllArgs] + Not + [StoreResult],
    ...
    'ooisnull':    [PushAllArgs, 'ldnull', 'ceq', StoreResult],
    'oononnull':  [PushAllArgs, 'ldnull', 'ceq'] + Not + [StoreResult],
    ...
    'int_add':     'add',
    'int_sub':     'sub',
    'int_mul':     'mul',
    'int_floordiv': 'div',
    'int_mod':     'rem',
    'int_lt':      'clt',
    ...
}

```

Figure 7.4: *Operations table for the static CLI backend*

7.4 Code generation

To generate the actual loop body, the backend needs to convert each operation of the intermediate representation used by the JIT into one or more CLI instructions. This intermediate representation directly derives from the one used by the translation toolchain (see Section 3.2): in particular, most of the operations have the same name and semantics.

By exploiting this fact, we can write the JIT backend very efficiently without need to teach it explicitly how to translate each operation: the CLI static backend⁸ already knows how to convert operations into CLI instructions, thus we just need a way to reuse this knowledge.

The CLI static backend contains a table mapping operations into CLI instructions: Figure 7.4 shows an excerpt of this table. We won't explain in detail its content, but from the figure it should be clear that it uses an internal language to express CLI instructions: in particular, every string such as `'add'`, `'sub'` or `'ldnull'` is translated directly into the corresponding CLI instruction, and moreover there are internal commands to do a more complex translation (e.g., `PushAllArgs` pushes all the arguments of the operation on the CLI stack).

Unfortunately, the CLI JIT backend cannot directly use this table for generating the code.

⁸It is important here not to confuse the CLI static and JIT backends: the first operates at translation time, and produces a text file containing the classes and the methods composing *pypy-cli*, which is then compiled into an executable. On the other hand, the JIT backend operates at runtime and generates CLI bytecode by invoking the API exposed by the `Reflection.Emit` namespace.

```

def emit_op_getfield_gc(self, op):
    descr = op.descr
    assert isinstance(descr, FieldDescr)
    clitype = descr.get_self_clitype()
    fieldinfo = descr.get_field_info()
    obj = op.args[0]

    # push the object on the stack, and maybe cast it to the correct type
    obj.load(self)
    if obj.getClitype(self) is not clitype:
        self.il.Emit(OpCodes.Castclass, clitype)

    # actually load the field value
    self.il.Emit(OpCodes.Ldfld, fieldinfo)

    # store the result in the proper variable
    self.store_result(op)

```

Figure 7.5: Implementation of the `getfield_gc` operation

As explained in Section 6.2 the JIT frontend and backends need to be written in RPython, while the operations table is not⁹. However, we can use the *metaprogramming* capabilities of RPython to automatically generate valid RPython code from the table.

Due to the way it is designed, the Translation Toolchain starts to analyze RPython modules only *after* they have been imported: this means that at import time we can use the full Python language to generate RPython classes and functions that will be later fed into the Translation Toolchain. In fact, we can think of Python as the metaprogramming language for RPython[AACM07].

The basic idea is turn each row in the table into a method that emits the corresponding CLI instructions: Figure 7.6 shows for example the method generated for the `int_add` operation, where `self.il` is an object of type `System.Reflection.Emit.ILGenerator` and it is used to emit the actual CLI bytecode.

```

def emit_op_int_add(self, op):
    self.push_all_args(op)
    self.il.Emit(OpCodes.Add)
    self.store_result(op)

```

Figure 7.6: Method generated for `int_add`

By exploiting this technique, we were able to considerably cut both the time needed to write the backend and the number of bugs, as the operation table was already well tested in the static CLI backend. However, there were more complex operations for which a simple translation from the operation table is not

⁹Remind that the Translation Toolchain does not need to be written in RPython, thus the static CLI backend can freely use the table.

enough: in those cases, we wrote the corresponding method manually, as for example it is the case of `getField_gc`, shown in Figure 7.5: in this example, it is also possible to see an usage of the `FieldDescr` descriptor introduced by Section 7.2.1.

7.5 Compiling inner bridges

Consider Figures 6.9 and 6.10, reported here for convenience as Figures 7.7 and 7.8. After the trace has been produced, the JIT frontend invokes the backend by calling `compile_loop`, which in turns generates a `.NET` method that contains the code of the loop. Later `guard_true(t1)` (highlighted in blue) becomes hot, and the JIT frontend produces a new trace, which is shown in green in Figure 7.8. At this point, the frontend calls `compile_bridge`.

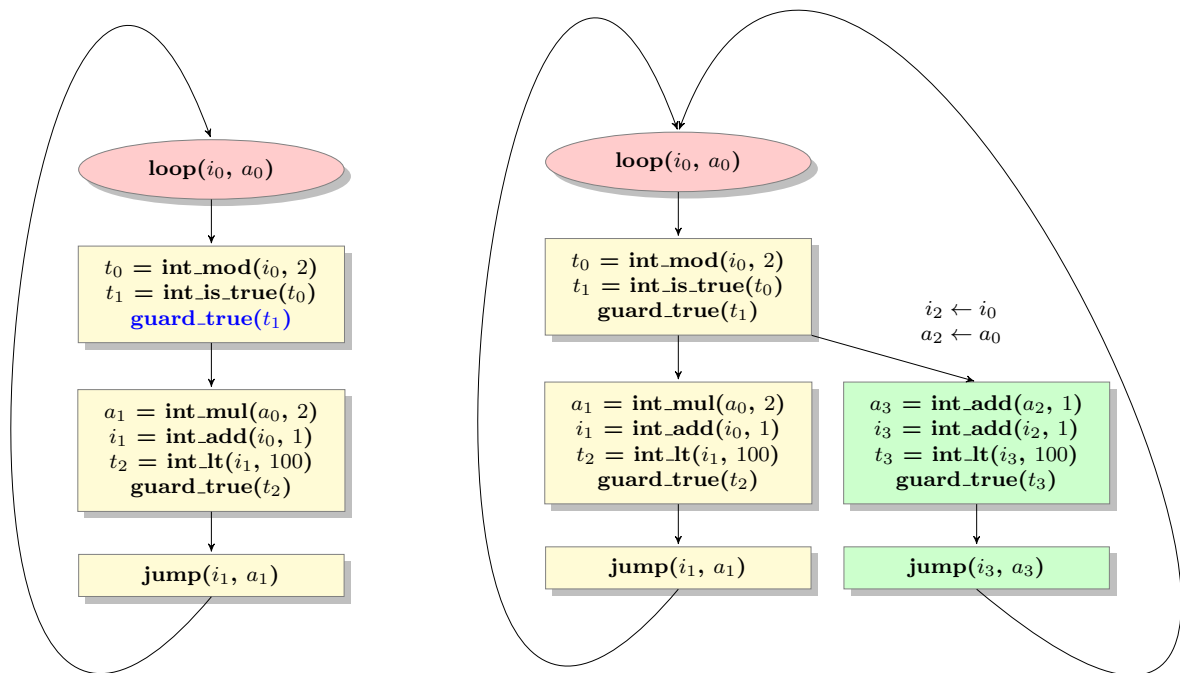


Figure 7.7: The trace produced by the code in Figure 6.8

Figure 7.8: The trace tree produced when the guard becomes hot

The *x86 backend* implements `compile_bridge` in a straightforward way: first, it emits the code for the new trace in memory, then it modifies the machine code for `guard_true(t1)` to jump to the newly generated bridge. However, this implementation strategy does not work with the CLI. As described in Section 4.3, `.NET` methods are immutable after they have

been compiled, thus, we cannot modify the code of the guard.

There are three main alternatives to solve the problem:

Recompiling the whole method Although we cannot modify the original method, we can always create a new one that contains the code for both the original loop and the new inner bridge. The code generated in this way is optimal, as we can directly use the fast `br` CLI instruction to jump from the failing guard to the bridge. However, this solution may imply that a considerable amount of time is wasted for re-compiling code: each time a new bridge is generated, the loop it belongs to must be recompiled together with all its current bridges. In the worst case, the time taken before having a stable loop (i.e. a loop that does no longer need to be recompiled because all its fast paths have been generated) is quadratic on the number of its paths

Using a *trampoline* The CLI does not offer the possibility of augmenting the code of a method after its compilation, but it is possible to simulate this behaviour. The idea is to have a *trampoline* that transfers the execution to a *child method*, which after execution returns a delegate (see Section 4.6) pointing to the next method to be called by the trampoline, and so on. The *next method* delegates returned by the children are not hard-coded in the bytecode, but stored in some place where they can be modified from the outside: later, we can alter the flow of control by modifying the *next method* delegate that is called when the guard fails to point to the newly created method that implements the bridge. The advantage of this technique is that we can compile new bridges without recompiling the existing loops, while the disadvantage is that it is extremely slow at runtime, because each control transfer involves a lot of operations. The author of this thesis has already explored this solution in [CAR09] and [ABCR08].

Using tail calls Tail calls are described in Section 4.4 and offer an alternative to the previous solution: instead of having the control flow bouncing between the trampoline and the children methods, we eliminate the former and directly transfer the execution from one method to the next one with a tail call. Again, the *next method* delegates are stored in some place where they can be modified, so that they can be edited when we compile a new bridge. In an ideal world, tail calls would be the perfect solution for us: they enable incremental compilation of bridges, and if implemented in the right way they are as efficient as local jumps. However, at the moment of writing they are too slow and too buggy to be used in practice when performance matters.

We chose to implement the first alternative, and to recompile the whole method every time a new bridge is generated. This way, the programs take more time to reach a stable state, but from that point on the code produced is as efficient as we can.

A better solution would consist in a mixed approach between the “recompile” approach and one of the other two outlined alternative solutions. At first, we could attach bridges using a trampoline or tail calls: then, when we see that the loop is “stable enough” and that no more fast path are discovered, we could recompile the whole loop into an efficient method. This approach has not been implemented yet, but we plan to experiment on it as a future work.

7.5.1 Entry and exit bridges

Both entry and exit bridges (see section 6.4) are executed only once, then they jump to another loop. Since they are created *after* the target loop, implementing them on the CLI poses the same problems as for inner bridges.

However, we chose to implement them differently: the rationale is that contrarily to inner bridges, which are supposed to be executed often and thus need to be implemented efficiently, entry and exit bridges are executed only once, then they transfer the execution to a loop which will probably run for a considerable amount of time. Hence, it is perfectly reasonable to use a slower implementation, if it has other advantages.

In particular, we implement jumps between bridges and loops as *tail calls*: this way we can avoid to recompile the whole loop tree every time we generate an entry or exit bridge that jumps to it.

Note that tail calls are necessary because it is possible to have loops that mutually jump to each other, as shown by Figure 7.9. In case we used simple calls instead of tail calls, we would exhaust the stack space after a while.

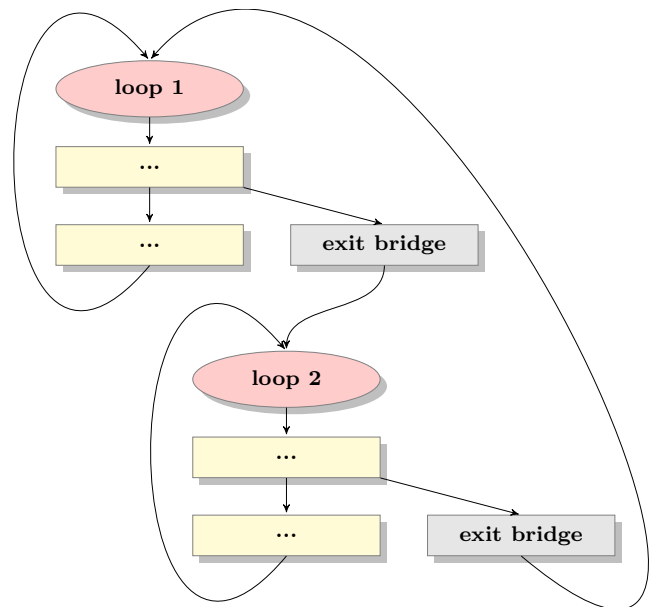


Figure 7.9: *Mutual recursive loops*

Chapter 8

Benchmarks

8.1 Applying the JIT to the Python Interpreter

Chapters 6 and 7 explain how the PyPy JIT generator works. In particular, in Section 6.2 we have shown that it is possible to generate a JIT compiler from every interpreter written in RPython. To measure the performance of the generated JIT, and in particular of the JIT with the CLI backend, we apply the JIT generator to the *Python Standard Interpreter* of PyPy (see Section 3.2).

As explained by Section 6.3.3, it is necessary to apply some *hints* to guide the JIT generation process. The two most important hints are `jit_merge_point` which indicates the start of the main interpreter loop, and `can_enter_jit` which indicates all the places where user loops have to be checked. Figure 8.1 shows a simplified version of the main interpreter loop of the Python Interpreter: note that `can_enter_jit` is called only at the end of `JUMP_ABSOLUTE`, which is the only Python opcode that can set the instruction pointer to an earlier value, i.e. the only one that can possibly lead to a user loop.

Apart from these two fundamental hints, the Python Interpreter contains about 30 more hints to guide the JIT to produce better code. These hints have not been explained in this thesis but, e.g., mark some classes as “immutable”, to let the JIT know that their instances never changes their state after they have been created. The amount of needed hints is quite modest compared to the about 100K LOC which compose the Python Interpreter: this supports the claim stated in Section 6.3.3 that the extra work needed to enable the JIT for a large interpreter is negligible.

Then, it is possible to invoke the translator and enable the JIT generator to get a final executable that contains both the Python Interpreter and the corresponding JIT compiler. In the following, we will refer to `pypy-cli` to indicate the Python interpreter translated

```

pypyjitdriver = JitDriver(greens = ['next_instr', 'pycode'],
                        reds = ['self', 'ec'])

class PyFrame(Frame):

    def dispatch(self, pycode, next_instr, ec):
        try:
            while True:
                pypyjitdriver.jit_merge_point()
                co_code = pycode.co_code
                next_instr = self.handle_bytecode(co_code,
                                                next_instr, ec)

        except ExitFrame:
            return self.popvalue()

    def JUMP_ABSOLUTE(self, jumpto, _, ec=None):
        self.last_instr = jumpto
        ec.bytecode_trace(self)
        jumpto = self.last_instr
        pypyjitdriver.can_enter_jit()
        return jumpto

    ...

```

Figure 8.1: *Main interpreter loop of the PyPy Python Interpreter*

with the CLI translation backend and the CLI JIT backend.

8.2 Making the interpreter JIT friendly

In order to get good performance it is necessary to have a very good trace optimizer (see section 6.5). In an ideal world, the optimizer would be smart enough to optimize all the possible programming patterns that are found in the interpreter. However, the reality is that at the moment not all programming patterns are optimized equally well, with the result that some code fragments are optimized by the JIT more than others.

The net result is that `pypy-cli` can occasionally encounter a user program that requires the execution of those interpreter fragments which cannot be reasonably optimized by the JIT, thus generating non optimal and sometimes even inefficient code. To solve these problems there are two possible solutions: either enhancing the optimizer, or rewriting the involved parts of the interpreter to allow the JIT to perform some useful optimization.

At the moment of writing the are known fragments of the Python Interpreter where the

JIT performs badly, as Section 8.4 highlights very well. These fragments include:

- Old-style classes¹: the PyPy team has always put more effort on optimizing new-style classes than old-style, with the result that the former are dramatically faster both during normal interpretation and with the JIT.
- String manipulation: some of the algorithms for string processing implemented in PyPy are inferior to the ones used by CPython or IronPython, thus even with the help of the JIT there are cases in which they perform badly.
- Regular expressions: in theory, it should be possible to consider the regular expression engine as another interpreter to be JITted separately from the main one, thus getting very optimized machine code for each different regular expression. However, at the moment this is not possible for technical reasons, and thus the regular expression engine is not considered by the JIT.

None of the above items represents a fundamental issue that cannot be solved by our approach, but have never been tackled due to time constraints. However at the moment of writing the PyPy team is actively working on them.

8.3 Methodology

To measure the performance of `pypy-cli` we selected two set of benchmarks: the first is a collection of 82 microbenchmarks which test various aspects of the Python language, while the second is a collection of 13 middle-sized programs that do various kind of “real life” computations.

The performance of `pypy-cli` is measured against IronPython. IronPython has a number of command-line options to enable or disable some Python features that are hard to implement efficiently. Since `pypy-cli` fully supports all of these features, we decided to enable them also in IronPython to make a fair comparison. In particular, we launched IronPython with the options `-X:FullFrames` which enables support for `sys._getframe`, and `-X:Tracing`, which enables support for `sys.settrace` (see Section 2.1.6 for more details). Note that the spirit behind the choices of the PyPy and IronPython teams is different: in IronPython, it is considered reasonable to disable such rarely used features by default because they lead to less efficient code, while the goal of PyPy is to implement the full semantics of the language without compromises, and let the JIT to optimize the code fragments where such

¹old-style classes implement the classical object model of Python. Since version 2.2, they have been deprecated and put side by side to *new-style* classes, which are slightly incompatible.

dynamic features are not used, while still producing correct results when they are actually used.

Following the methodology proposed by [GBE07], each microbenchmark and middle-sized benchmark has been run for 10 and 30 times in a row, respectively. Since we are interested in steady-state performance the first run of each benchmark has been discarded, as it supposedly includes the time spent by all the layers of JIT compilation. The final numbers were reached by computing the average of all other runs, the confidence intervals were computed using a 95% confidence level.

All tests have been performed on an otherwise idle machine with an *Intel Core i7 920* CPU running at 2.67 GHz with 4GB RAM. All benchmarks were run both under Linux using Mono, the open source implementation of the CLI, and under Microsoft Windows XP using the CLR, which is the original implementation of the CLI by Microsoft. The following is a list of the versions of the software used:

- Ubuntu 9.10 *Karmic Koala*, with Linux 2.6.31
- Mono 2.4.2.3
- IronPython 2.6.10920.0
- Microsoft Windows XP SP2
- Microsoft CLR 2.0.50727.1433²

8.4 Microbenchmarks

Figure 8.2 shows the results of the microbenchmarks: for each benchmark, the bar indicates the speedup (or slowdown) factor of `pypy-cli` compared to IronPython.

On Mono `pypy-cli` performs very well on most benchmarks with speedups up to 215 times faster and only 20 ones slowed down, as summarized by Figure 8.3.

On CLR `pypy-cli` performs a bit worse, but the results are still very satisfactory, with speedups up to 155 times faster and only 21 benchmarks slowed down. However, in some cases the slowdown is very considerable, up to 78 times slower. Figure 8.4 summarizes the results.

²At the moment of writing, the most recent version of the .NET Framework is 3.5. However the virtual machine at the core of the Framework has not been updated since .NET 2.0, hence the version of the CLR.

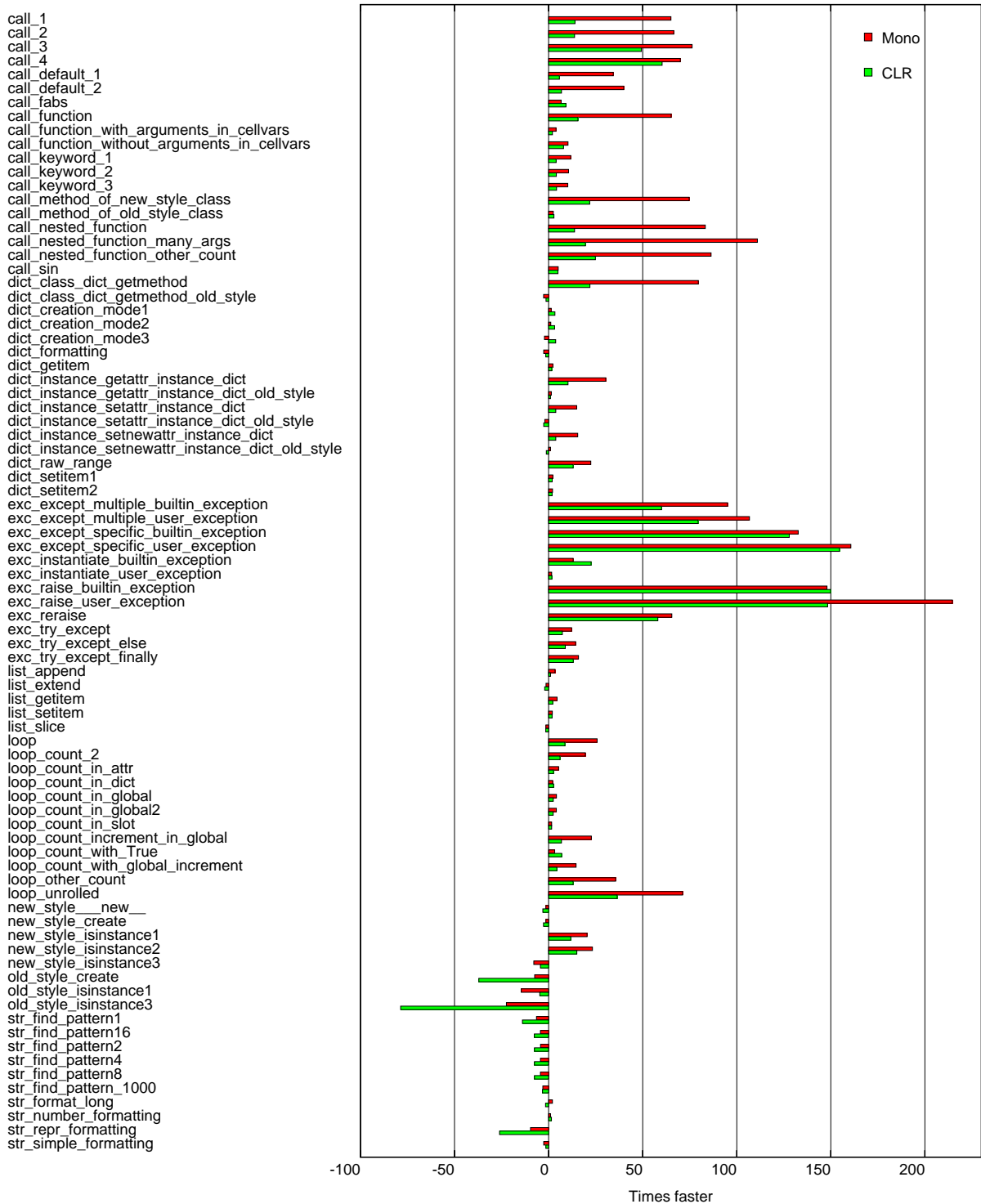


Figure 8.2: Speedup of pypy-cli vs IronPython (microbench)

Speedup factor				No.
100	to	250	faster	6
25	to	100		16
10	to	25		17
2	to	10		15
1	to	2		8
1	to	2	slower	4
2	to	10		14
10	to	25		2
25	to	100		0

Figure 8.3: *Microbenchmarks on Mono*

Speedup factor				No.
100	to	250	faster	4
25	to	100		6
10	to	25		15
2	to	10		27
1	to	2		9
1	to	2	slower	6
2	to	10		11
10	to	25		2
25	to	100		2

Figure 8.4: *Microbenchmarks on CLR*

It is interesting to note that most of the bad results are related to the execution of those fragments where the JIT is known to perform badly, as described in Section 8.2: in particular, old-style classes and operations that involve strings. Moreover, it also emerges that all the microbenchmarks that create a massive amount of new objects are usually slower. Another noteworthy observation is that generally `pypy-cli` has a greater speedup factor over IronPython on Mono than on CLR.

8.5 Middle-sized benchmarks

Figure 8.5 contains a short descriptions of the 13 middle-sized benchmarks that have been executed to further evaluate the performance of `pypy-cli` versus IronPython.

Figure 8.6 shows the time taken by `pypy-cli` and IronPython to complete each benchmark. The results on Mono show clearly that for the major part of the benchmarks, `pypy-cli` is faster than IronPython. It is interesting to analyze the benchmarks that are slower:

- `build-tree` creates a massive amount of new objects, showing the same behaviour already noted for the microbenchmarks. This is an unexpected result that have not been investigated further due to time constraints, but will be analyzed as a part of the future work.
- `fannkuch` does not even complete. This is because the JIT creates two *mutual recursive loops* (see Section 7.5.1) that calls each other with a tail-call. However, due to a bug in the implementation of tail calls on Mono (see Section 4.4) the program exhausts the stack space and terminates abnormally.

Name	Description
build-tree	Create and traverse of a huge binary tree (1 million of elements)
chaos	Generate an image containing a chaos game-like fractal
f1	Two nested loops doing intensive computation with integers
fannkuch	The classical <i>fannkuch</i> benchmark [AR]
float	Exercise both floating point and object oriented operations by encapsulating (x, y) pairs in a <code>Point</code> class
html	Generate a huge HTML table
k-nucleotide	Analyze nucleotide sequences expressed in <i>FASTA format</i> [fBI10]
linked-list	Create a huge linked list
oobench	Measure the performance of method invocation on objects
pystone	The standard Python benchmark, derived from the classical <i>Dhrystone</i> [Wik10a]
pystone-new	The same as <code>pystone</code> , but using new-style classes instead of old-style ones
richards	The classical <i>Richards</i> benchmarks, originally written in BCPL and rewritten in Python
spectral-norm	Compute the eigenvalue using the power method

Figure 8.5: *Description of the benchmarks*

- `html` involves a lot of operations on strings, thus `pypy-cli` was expected to be slower. However it is interesting to see that the difference is minimal: this is probably due to the fact that all the other operations (like method invocation and attribute access) are speeded up by `pypy-cli`, balancing the slowdown caused by the string operations.
- `k-nucleotide` is only about operations on strings, thus the slowdown is in line with what observed by the microbenchmarks.
- `pystone` is slower because it uses old-style classes. `pystone-new`, which uses new-style classes, is much faster on `pypy-cli`.

On CLR the results are less satisfactory: 7 out of 13 benchmarks are faster, but the other 6 are slower, sometimes even by a large factor. It is interesting to note that on the two benchmarks that involve a lot of string manipulation (`html` and `k-nucleotide`), `pypy-cli` behaves better on CLR than on Mono and outperforms IronPython. In the CLR histogram, the data for `chaos` have been omitted for `pypy-cli` because it produces incorrect results: this is probably due to a bug in the CLR itself, as on Mono it works correctly.

The histogram in Figure 8.7 summarizes the results by showing the speedup (or slowdown) factor of `pypy-cli` over IronPython. `f1` and `oobench` are the benchmarks with the largest

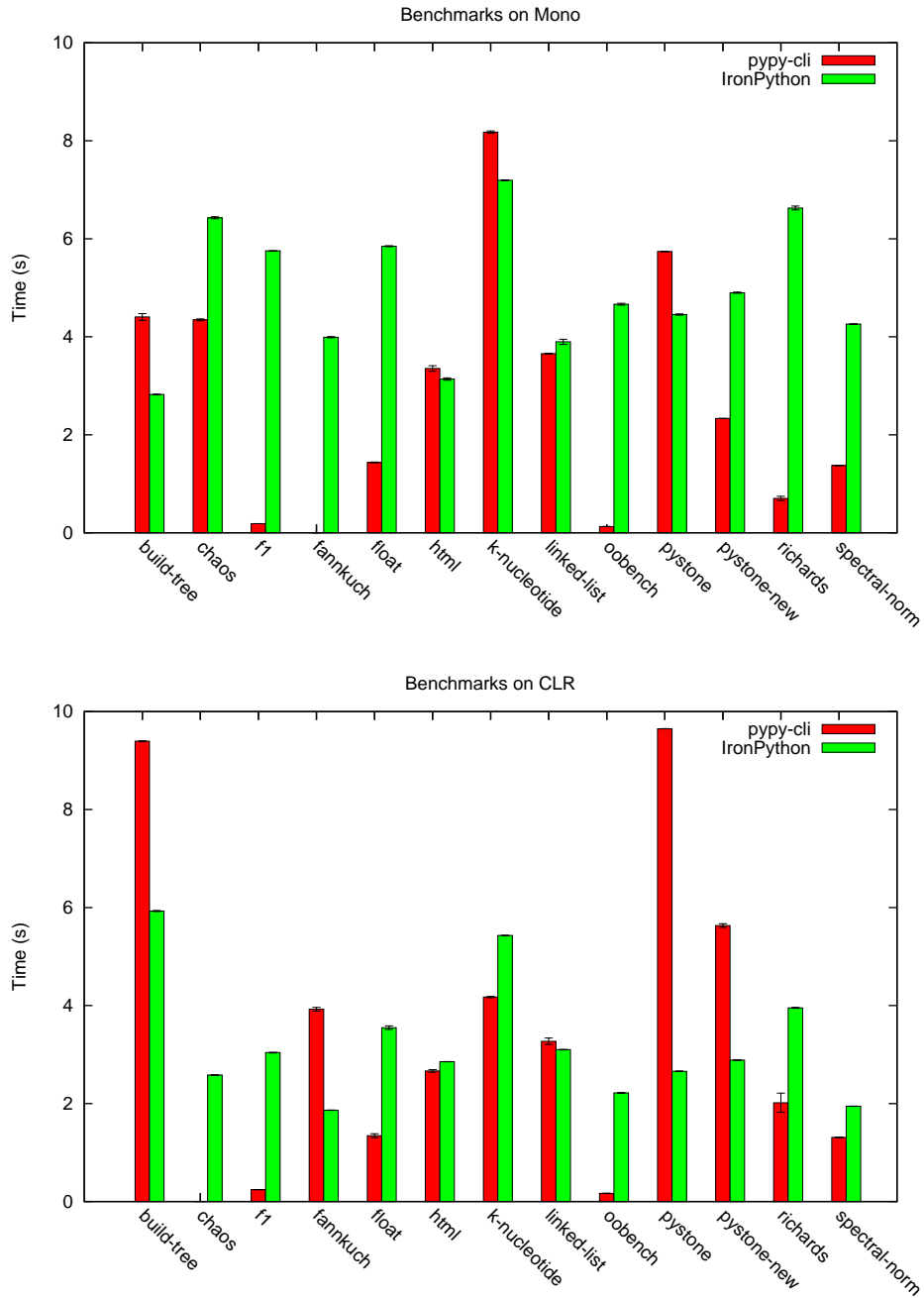


Figure 8.6: Time taken to complete the benchmarks on Mono and CLR (the lower the better)

speedup factor, up to about 31 and 37 times faster on Mono, and 12 and 13 times faster on CLR. `float`, `richards` and `spectral-norm` are also consistently speeded up but by smaller factor. Some benchmarks (`html`, `k-nucleotide`, `linked-list`, `pystone-new`) exhibit a controversial behavior, as they are speeded up by an implementation and slowed down by the other, or vice versa. Finally, `build-tree` and `pystone` are consistently slowed down by `pypy-cli`.

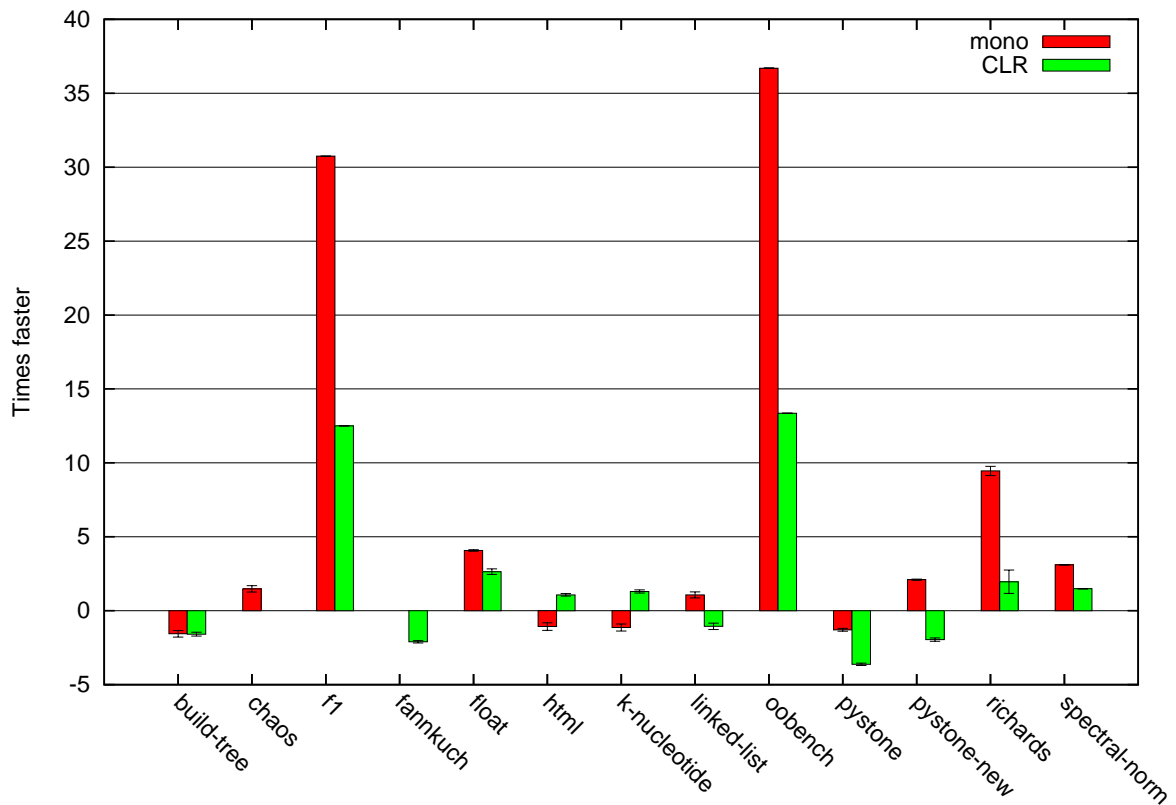


Figure 8.7: Speedup factor of `pypy-cli` over IronPython on Mono and CLR (the higher the better for `pypy-cli`)

8.5.1 Differences between Mono and CLR

Why on Mono `pypy-cli` gets much better results compared to IronPython? Figure 8.8 compares side-by-side the performance of the two platforms when running either `pypy-cli` or IronPython: remind that all the benchmarks were run on the same machine (although on different operating systems), so the results are directly comparable. With the sole

exception of `build-tree`, IronPython is consistently faster on CLR than on Mono. This is probably due to the fact that IronPython has been explicitly optimized for the CLR and vice versa, since both projects have been developed by Microsoft.

On the other hand, for some benchmarks `pypy-cli` behaves better on Mono than on CLR, while for others is the opposite. Probably, this is due to the fact that the bytecode generated by the higher level JIT compiler of `pypy-cli` is very different from the typical bytecode generated by other compilers for CLI (e.g. by C# compilers), thus is not always properly optimized by the lower level JIT compiler. The fact that `chaos` does not work correctly on CLR supports this theory, as the code produced by `pypy-cli` triggers a bug that it has probably been unobserved for years.

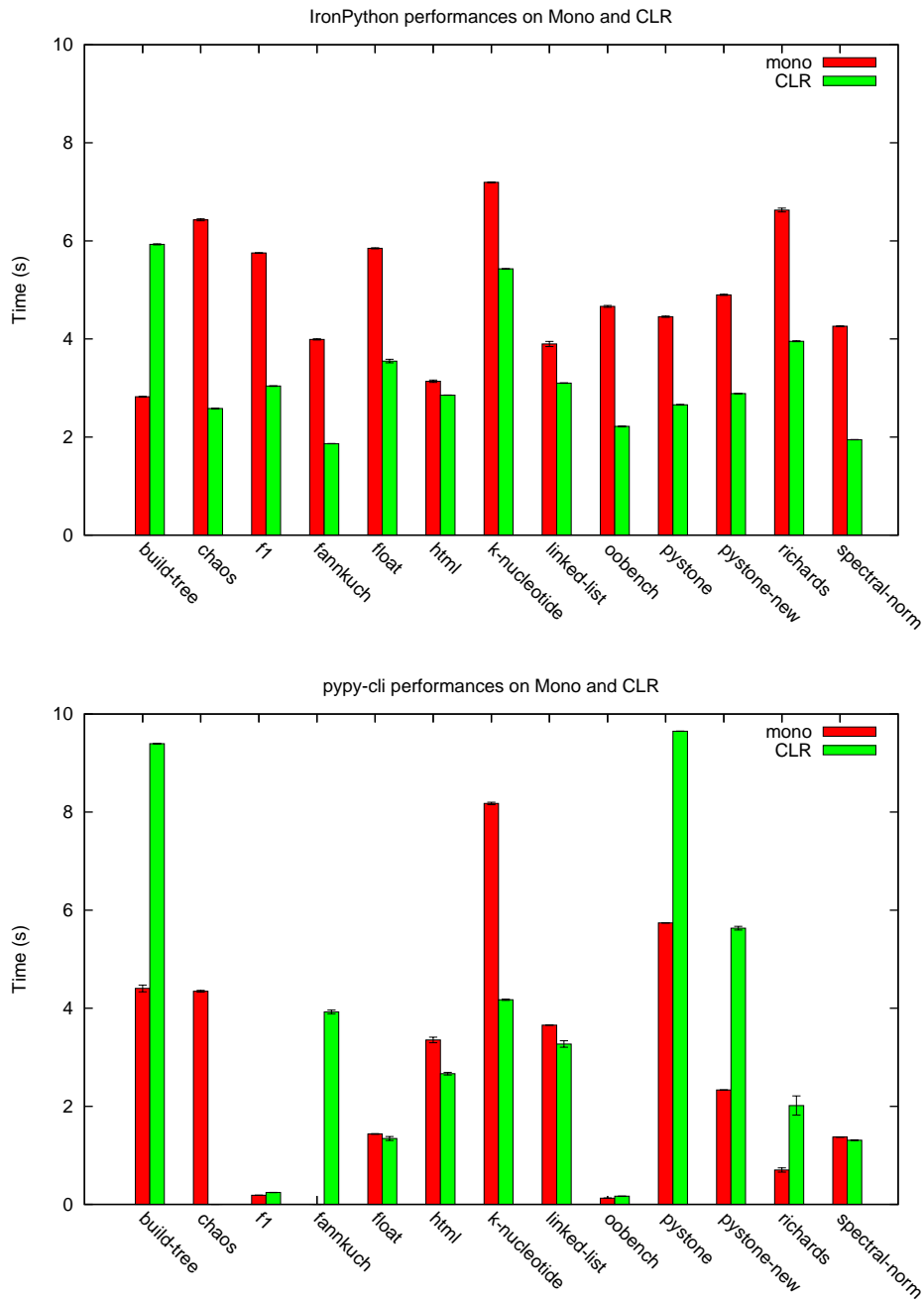


Figure 8.8: Time taken to complete the benchmarks by `pypy-cli` and `IronPython` (the lower the better)

Chapter 9

Conclusion and Future Work

9.1 Contributions of this thesis

This thesis investigates new techniques to implement in an efficient and effective way dynamic languages on top of object oriented, statically typed virtual machines. Concretely, these techniques have been applied to Python as an example of a complex, real world and extremely dynamic programming language, and to the CLI as the chosen target virtual machine. However, most of the developed concepts and of the experience gathered in this thesis can be effectively reused for other dynamic languages and target virtual machines, in particular the JVM.

One of the most important contribution is the introduction of the concept of *JIT layering* (see Section 4.2), which introduces a higher level of JIT compilation on top of the existing JIT compiler of the virtual machine.

However neither the CLI nor the JVM have been designed to be a target for a JIT compiler, and they lack features that are fundamental to emit highly efficient code (see e.g. Sections 4.3 and 4.4). Sections 7.3, 7.5 and 7.5.1 explain how to overcome these limitations and what are the trade-offs for the different solutions.

Instead of using a more traditional JIT compilation strategy, we implemented a *tracing JIT* (see Chapter 5). To our knowledge, this is the first example of a tracing JIT compiler targeting the CLI or similar virtual machines.

Instead of directly implementing a tracing JIT, we have developed a *tracing JIT generator* that can be applied to any interpreter written in RPython. Interpreters offer the simplest and cheapest way to implement a language, however compilers allow a much more efficient implementation at the cost of a more expensive development and maintenance effort; furthermore, such drawbacks are even more serious for JIT compilers. By automatically

generating a JIT from an interpreter we try to take the best of both approaches, as explained in Chapter 6. The PyPy JIT generator is the result of the development efforts of the PyPy team to which the author of this thesis has actively collaborated.

The CLI JIT backend, described in Chapter 7, has been entirely developed by the author of this thesis. One of the most challenging problems which has been faced during the implementation of the backend regards JIT compilation of fragments of the same method at different times, since the CLI does not directly support incremental JIT compilations of methods.

Finally, a benchmark suite has been executed to evaluate the performance of the JIT compiler (called `pypy-cli`) produced by the PyPy JIT generator from the PyPy Python interpreter. Such benchmarks have been executed both with `pypy-cli` and with IronPython, and on top of the two most widespread and stable available implementations of CLI, that is Mono (on Linux) and CLR (on Microsoft .NET).

Overall the results are satisfactory: most benchmarks run faster on `pypy-cli` than on IronPython, up to 40 times faster: this proves that, at least for the CLI and for some kinds of computations, the technique of *JIT layering* works well and that a *tracing JIT compiler* can lead to dramatically better results than a traditional static compiler like IronPython.

Nevertheless, if on the one hand some benchmarks are considerably speeded up by `pypy-cli`, on the other hand some others are slowed down; moreover, some benchmarks are speeded up on Mono but slowed down on CLR or vice versa. This fact validates the intuitive assumption that the final performance is equally influenced by both the high and low level JIT compilers. However, what emerges is that it is hard if not impossible to predict the performance of a particular program before running it.

Finally, one benchmark did not complete the execution under Mono because of a bug in the Virtual Machine. Similarly, one benchmark gave the wrong results when run under CLR: we did not investigate further the source of the misbehaviour, which could be perhaps due to a bug of the Virtual Machine. It is interesting to note that we managed to hit two different bugs running only a limited set of benchmarks despite the fact that both implementations are considered very stable and of high quality, and widely used in production environments. This is probably due to the fact that our tracing JIT compiler produces patterns of code that are different from the typical patterns of code generated by the compilers for more widespread languages such as C#, but is also indicates that both implementations are not well tested enough.

In conclusion, we can say that the concepts and the techniques presented in this thesis give very good results in some cases and are thus worth of further investigation and development.

9.2 Future work

There are many different directions for future work related with this thesis.

First of all, we want to analyze more in depth the behavior of `pypy-cli` on the benchmarks where it is slower than IronPython to see if it is possible to further improve the overall performance of the system. Moreover, it would be interesting to analyze the reasons why some benchmarks are faster on one implementation of the CLI but not on the other, to see if it is possible to make `pypy-cli` consistently faster than IronPython.

Another problem of the current implementation of the CLI JIT backend is the relatively high cost of the JIT compilation: Section 7.5 describes in detail the problem as well as a possible solution.

Finally, it would be interesting to apply the JIT generator to other dynamic languages written in RPython, to demonstrate that PyPy can be an effective framework to implement highly efficient programming languages for the CLI. Moreover, we want to port the JIT generator to the JVM. We described how the techniques presented in this thesis are not currently applicable to the CLI in production environments due to the instability and immaturity of its implementations, but it would be interesting to see how they behaves on the JVM, whose implementations are generally more mature, notably *HotSpot*. Moreover, there *Da Vinci Machine Project*[mlv] is actively researching how to extend *HotSpot* with new features thought to implement dynamic languages on the JVM, which could probably be exploited to implement the JVM JIT backend for PyPy.

Bibliography

- [AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64, Montreal, Quebec, Canada, 2007. ACM.
- [ABCR08] Davide Ancona, Carl Friedrich Bolz, Antonio Cuni, and Armin Rigo. Automatic generation of JIT compilers for dynamic languages in .NET. Technical report, DISI, University of Genova and Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, 2008.
- [AFG⁺05] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.
- [Ana99] C. Scott Ananian. The static single information form. Technical Report MIT-LCS-TR-801, MIT Laboratory for Computer Science Technical Report, September 1999. Master’s thesis.
- [AR] Kenneth Anderson and Duane Rettig. Performing lisp analysis of the fannkuch benchmark.
- [asm] Asm - home page. <http://asm.ow2.org/>.
- [bce] The byte code engineering library. <http://jakarta.apache.org/bcel/>.
- [BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy’s tracing jit compiler. In *ICOOOLPS ’09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM.

- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
- [BKL⁺08] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. 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, First Workshop, S3 2008, Potsdam, Revised Selected Papers*, volume 5146 of *Lecture Notes in Computer Science*, pages 123–139, 2008.
- [Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to java. In *In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 20–34, 1999.
- [BLR09] Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo. Towards just-in-time partial evaluation of prolog. In Danny De Schreye, editor, *LOPSTR*, volume 6037 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2009.
- [bug] Bug 502575 - Tail call problems with F#. https://bugzilla.novell.com/show_bug.cgi?id=502575.
- [BV09] Camillo Bruni and Toon Verwaest. Pygirl: Generating whole-system vms from high-level prototypes using pypy. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 328–347. Springer, 2009.
- [Can05] Brett Cannon. Localized type inference of atomic types in python, 2005.
- [CAR09] Antonio Cuni, Davide Ancona, and Armin Rigo. Faster than c#: efficient implementation of dynamic languages on .net. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 26–33, New York, NY, USA, 2009. ACM.
- [CBY⁺07] Mason Chang, Michael Bebenita, Alexander Yermolovich, Andreas Gal, and Michael Franz. Efficient Just-In-Time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

- [CPC⁺07] Antonio Cuni, Samuele Pedroni, Anders Chrigström, Holger Krekel, Guido Wesdorp, and Carl Friedrich Bolz. High-level backends and interpreter feature prototypes. Technical Report D12.1, PyPy Consortium, 2007. <http://codespeak.net/pypy/dist/pypy/doc/index-report.html>.
- [CSR⁺09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for Web 3.0: Trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 71–80, Washington, DC, USA, 2009. ACM.
- [Cun06] Antonio Cuni. Implementing python in .NET. Technical report, DISI, University of Genova, 2006. http://codespeak.net/~antocuni/papers/implementing_python_in_dotnet2006.pdf.
- [dCGS⁺99] Jong deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *In Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–19. ACM Press, 1999.
- [dot] Microsoft .NET Framework.
- [EG03] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.
- [fBI10] National Center for Biotechnology Information. Fasta format description, 2010. <http://www.ncbi.nlm.nih.gov/blast/fasta.shtml>.
- [GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–76. ACM, 2007.
- [GES⁺09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Michael Bebenita, Mason Chang, Michael Franz, Edwin Smith, Rick Reitmaier, and Mohammad Haghghat. Trace-based Just-in-Time type specialization for dynamic languages. In *PLDI*, 2009.
- [GF06] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, November 2006.
- [GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design, 1993.

- [GPF06] Andreas Gal, Christian W. Probst, and Michael 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.
- [H94] Urs Hölzle. Adaptive optimization for SELF: reconciling high performance with exploratory programming. Technical report, Stanford University, 1994.
- [HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag.
- [HH93] Philipp Hoschka and Christian Huitema. Control flow graph analysis for automatic fast path implementation. In *In Second IEEE workshop on the architecture and Implementation of*, 1993.
- [Int06] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, June 2006.
- [iro] IronPython. <http://www.codeplex.com/IronPython>.
- [jyt] The jython project. <http://jython.org>.
- [LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO201904)*, 2004.
- [mlv] the Da Vinci Machine Project. <http://openjdk.java.net/projects/mlvm/>.
- [PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot(tm) server compiler. In *In USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.
- [pyp] Pypy. <http://codespeak.net/pypy>.
- [pyt] Python programming language. <http://www.python.org>.
- [Rig04] Armin Rigo. Representation-based just-in-time specialization and the Psycho 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.
- [RP06] A. Rigo and S. Pedroni. PyPy’s approach to virtual machine construction. In *OOPSLA Companion*, pages 944–953, 2006.

- [Sal04] Michael Salib. Faster than c: Static type inference with starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.
- [Sch09] Arnold Schwaighofer. Tail call optimization in the java hotspot(tm) vm, 2009.
- [SPH01] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Simple generators, 2001. <http://www.python.org/dev/peps/pep-0255/>.
- [Ste77] Guy Lewis Steele, Jr. Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *ACM '77: Proceedings of the 1977 annual conference*, pages 153–162, New York, NY, USA, 1977. ACM.
- [unl] unladen-swallow. <http://code.google.com/p/unladen-swallow/>.
- [Wik10a] Wikipedia. Dhrystone — wikipedia, the free encyclopedia, 2010. <http://en.wikipedia.org/wiki/Dhrystone>.
- [Wik10b] Wikipedia. Pareto principle — wikipedia, the free encyclopedia, 2010. http://en.wikipedia.org/wiki/Pareto_principle.

