IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

D08.1 Release a JIT Compiler for PyPy Including Processor Backends for Intel and PowerPC

Due date of deliverable: 31st March, 2007

Actual Submission date: 2nd May, 2007

Start date of Project: 1st December 2004 **Duration: 28 months**

Lead Contractor of this WP: Strakt

Authors: Samuele Pedroni (Strakt), Armin Rigo (HHU)

Revision: Final

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
2007-04-26	Samuele Pedroni	contents
2007-04-30	Carl Friedrich Bolz	publish final version on the web page

Abstract and Executive Summary

PyPy 1.0 was released on March 27th 2007, it delivers Just-In-Time compilation capabilities to our generated Python interpreters and machine code backends for both IA32 and the PowerPC. The Just-In-Time compiler is generated starting from PyPy's high-level implementation of Python through our translation framework. This represents a remarkable result and was one of our initial scientific objectives. This dynamic compiler generation technology can be generally applied to other languages implemented using our translation framework.

Purpose, Scope and Related Documents

This document reports on our successful delivery of a Python Just-In-Time compiler in our PyPy 1.0 release, together with the technology to generate it.

For the details of our dynamic compiler generation approach and its relevance see:

- D08.2: JIT Compiler Architecture

For other aspects and features delivered with 1.0 see:

- D12.1: High-Level Backends and Interpreter Feature Prototypes
- D13.1: Integration and Configuration
- D14.4: PyPy-1.0 Milestone report

Contents

Appendices

1 PyPy Python JIT and JIT Generation framework in PyPy 1.0

We planned since the beginning for the last milestone of the project to have a Python implementation which included a Just-In-Time compiler. It was one of our major research goal for this dynamic compiler not to be written but instead generated through one further transformation step in our translation framework ((D05.1), (VMC)). Both these objectives have been essentially achieved. PyPy 1.0 released on contains March 27th 2007 a translation step able to produce a dynamic compiler based on hints added to the codebase of the translated interpreter. This process is not specific to our Python interpreter and should be generally applicable to any interpreter written in RPython to be translated by our framework.

The quality and performance of the results of such generated dynamic compiler depends on which parts of the interpreter are transformed to become compilers for the functionality they implement, and the effective application of hints.

The hints are annotations about which runtime information is valuable to feedback into the compiler to produce code optimized for values of that information actually occurring at runtime. Types are a typical example of such information that is valuable to feedback. For details see our report on the architecture and principle of this transformation (D08.2) and the Appendix 1 attached to this report.

In PyPy 1.0 enough hints were added and enough of the Python interpreter is transformed such that at least integer operations can be dynamically compiled into efficient code and dispatch loop overhead is removed. We have simple benchmarks that demonstrate this (D08.2) .

This result validates our approach and efforts, although more extensive hints are necessary to have more generalized speed-ups. Their addition is going to be part of work to be done after the project.

By construction, the JIT should work correctly on absolutely any kind of Python code: generators, nested scopes, `exec` statements, `sys._getframe().f_back.f_back.f_locals`, etc. (the latter is an example of expression that no existing Python or Python-like compiler emulates correctly). See again (D08.2) for the approach by which this is achieved.

In PyPy 1.0 the JIT framework contains two machine code backends, one for IA32 and one for the PowerPC (PPC) . This means that low-level translated PyPy Python interpreters can be produced capable of machine code generation for these two processor architectures. The dynamic compilers are fairly OS-independent and have been tested on top of Linux, Mac OS/X (both Intel and PPC) and Windows.

2 How to compile a pypy-c with a JIT

Go to `pypy/translator/goal/` in the PyPy source code distribution and run:

```
./translate.py --jit targetpypystandalone
```

This will produce the C code for a version pypy-c that includes both a regular interpreter and an automatically generated JIT compiler. This pypy-c uses its interpreter by default, and due to some overhead we expect this interpreter to be a bit slower than the one found in a pypy-c compiled without JIT.

In addition to `--jit`, it is possible to also pass the normal options to `translate.py` to compile different flavors of PyPy with a JIT. See the compatibility matrix in (D13.1) for the combinations

known to be working right now. (The combination of the JIT with the thunk or taint object spaces probably works too, but we don't expect it to generate good code before we add some extra hints in the source code of the object spaces.)

3 Usage

One or many code objects can be marked as candidates for being run by the JIT as follows:

```
>>>> def f(x): return x*5
>>>> import pypyjit
>>>> pypyjit.enable(f.func_code)
>>>> f(7)
# the JIT runs here
35
>>>> f(8)
# machine code already generated, no more jitting occurs here
40
```

A few examples of this kind can be found in `demo/jit/`. Although the JIT generation process is well-tested, we only have a few tests directly for the final `pypy-c`. Try:

```
pypy-c test_all.py module/pypyjit/test/test_pypy_c.py -A --nomagic
```

One can get a dump of the generated machine code by setting the environment variable `PYPYJITLOG` to a file name before pypy-c is started. To inspect this file, use the following tool:

```
python  pypy/jit/codegen/i386/viewcode.py  dumpfilename
```

The viewcode.py script works on Linux and is based on the `objdump` tool to produce a disassembly.

4 Documentation

Initial documentation on how to write interpreters with included dynamic compilation features can be found in this document as Appendix 1.

5 Glossary of Abbreviations

The following abbreviations may be used within this document:

5.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree

CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple Direct-Media Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

5.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

(D05.1) *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

(D05.4) *Encapsulating Low-Level Aspects*, PyPy EU-Report, 2005

(VMC) *PyPy's approach to virtual machine construction*, Armin Rigo, Samuele Pedroni, in OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications, pp. 944-953, ACM Press, 2006

(D08.2) *JIT Compiler Architecture*, PyPy EU-Report, 2007

(D13.1) *Integration and Configuration*, PyPy EU-Report, 2007

Make your own JIT compiler

Introduction

The central idea of the PyPy JIT is to be *completely independent from the Python language*. We did not write down a JIT compiler by hand. Instead, we generate it anew during each translation of pypy-c.

This means that the same technique works out of the box for any other language for which we have an interpreter written in RPython. The technique works on interpreters of any size, from tiny to PyPy.

Aside from the obvious advantage, it means that we can show all the basic ideas of the technique on a tiny interpreter. The fact that we have done the same on the whole of PyPy shows that the approach scales well. So we will follow in the sequel the example of small interpreters and insert a JIT compiler into them during translation.

The important terms:

- *Translation time*: while you are running `translate.py` to produce the static executable.
- *Compile time*: when the JIT compiler runs. This really occurs at runtime, as it is a Just-In-Time compiler, but we need a consistent way of naming the part of runtime that is occupied by running the JIT support code and generating more machine code.
- *Run time*: the execution of the user program. This can mean either when the interpreter runs (for parts of the user program that have not been JIT-compiled), or when the generated machine code runs.

A first example

Source

Let's consider a very small interpreter-like example:

```
def ll_plus_minus(s, x, y):
    acc = x
    pc = 0
    while pc < len(s):
        op = s[pc]
        hint(op, concrete=True)
        if op == '+':
            acc += y
        elif op == '-':
            acc -= y
        pc += 1
    return acc
```

Here, `s` is an input program which is simply a string of `'+'` or `'-'`. The `x` and `y` are integer input arguments. The source code of this example is in pypy/jit/tl/tiny1.py.

Hint

Ideally, turning an interpreter into a JIT compiler is only a matter of adding a few hints. In practice, the current JIT generation framework has many limitations and rough edges requiring workarounds. On the above example, though, it works out of the box. We only need one hint, the central hint that all interpreters need. In the source, it is the line:

```
hint(op, concrete=True)
```

This hint says: "at this point in time, ensure that `op` is a compile-time constant". The motivation for such a hint is that the most important source of inefficiency in a small interpreter is the switch on the next opcode, here `op`. If `op` is known at the time where the JIT compiler runs, then the whole switch dispatch can be constant-folded away; only the case that applies remains.

The way the `concrete=True` hint works is by setting a constraint: it requires `op` to be a compile-time constant. During translation, a phase called *hint-annotation* processes these hints and tries to satisfy the constraints. Without further hints, the only way that `op` could be a compile-time constant is if all the other values that `op` depends on are also compile-time constants. So the hint-annotator will also mark `s` and `pc` as compile-time constants.

Colors

You can see the results of the hint-annotator with the following commands:

```
cd pypy/jit/tl
python ../../translator/goal/translate.py --hintannotate targettiny1.py
```

Click on `ll_plus_minus` in the Pygame viewer to get a nicely colored graph of that function. The graph contains the low-level operations produced by RTyping. The *green* variables are the ones that have been forced to be compile-time constants by the hint-annotator. The *red* variables are the ones that will generally not be compile-time constants, although the JIT compiler is also able to do constant propagation of red variables if they contain compile-time constants in the first place.

In this example, when the JIT runs, it generates machine code that is simply a list of additions and subtractions, as indicated by the `'+'` and `'-'` characters in the input string. To understand why it does this, consider the colored graph of `ll_plus_minus` more closely. A way to understand this graph is to consider that it is no longer the graph of the `ll_plus_minus` interpreter, but really the graph of the JIT compiler itself. All operations involving only green variables will be performed by the JIT compiler at compile-time. In this case, the whole looping code only involves the green variables `s` and `pc`, so the JIT compiler itself will loop over all the opcodes of the bytecode string `s`, fetch the characters and do the switch on them. The only operations involving red variables are the `int_add` and `int_sub` operations in the implementation of the `'+'` and `'-'` opcodes respectively. These are the operations that will be generated as machine code by the JIT.

Over-simplifying, we can say that at the end of translation, in the actual implementation of the JIT compiler, operations involving only green variables are kept unchanged, and operations involving red variables have been replaced by calls to helpers. These helpers contain the logic to generate a copy of the original operation, as machine code, directly into memory.

Translating

Now try translating `tiny1.py` with a JIT without stopping at the hint-annotation viewer:

```
python ../../translator/goal/translate.py --jit targettiny1.py
```

Test it:

```
./targettiny1-c +++-+++ 100 10
150
```

What occurred here is that the colored graph seen above was turned into a JIT compiler, and the original `ll_plus_minus` function was patched. Whenever that function is called from the rest of the program (in this case, from `entry_point()` in pypy/jit/tl/targettiny1.py), then instead of the original code performing the interpretation, the patched function performs the following operations:

- It looks up the value of its green argument `s` in a cache (the red `x` and `y` are not considered here).
- If the cache does not contain a corresponding entry, the JIT compiler is called to produce machine code. At this point, we pass to the JIT compiler the value of `s` as a compile-time constant, but `x` and `y` remain variables.
- Finally, the machine code (either just produced or retrieved from the cache) is invoked with the actual values of `x` and `y`.

The idea is that interpreting the same bytecode over and over again with different values of `x` and `y` should be the fast path: the compilation step is only required the first time.

On 386-compatible processors running Linux, you can inspect the generated machine code as follows:

```
PYPYJITLOG=log ./targettiny1-c +++-+++ 100 10
python ../../jit/codegen/i386/viewcode.py log
```

If you are familiar with GNU-style 386 assembler, you will notice that the code is a single block with no jump, containing the three additions, the subtraction, and the three further additions. The machine code is not particularly optimal in this example because all the values are input arguments of the function, so they are reloaded and stored back in the stack at every operation. The current backend tends to use registers in a (slightly) more reasonable way on more complicated examples.

A slightly less tiny interpreter

The interpreter in pypy/jit/tl/tiny2.py is a reasonably good example of the difficulties that we meet when scaling up this approach, and how we solve them - or work around them. For more details, see the comments in the source code. With more work on the JIT generator, we hope to be eventually able to remove the need for the workarounds.

Promotion

The most powerful hint introduced in this example is `promote=True`. It is applied to a value that is usually not a compile-time constant, but which we would like to become a compile-time constant "just in time". Its meaning is to instruct the JIT compiler to stop compiling at this point, wait until the runtime actually reaches that point, grab the value that arrived here at runtime, and go on compiling with the value now considered as a compile-time constant. If the same point is reached at runtime several times with several different values, the compiler will produce one code path for each, with a switch in the generated code. This is a process that is never "finished": in general, new values can always show up later during runtime, causing more code paths to be compiled and the switch in the generated code to be extended.

Promotion is the essential new feature introduced in PyPy when compared to existing partial evaluation techniques (it was actually first introduced in Psyco [JITSPEC], which is strictly speaking not a partial evaluator).

Another way to understand the effect of promotion is to consider it as a complement to the `concrete=True` hint. The latter tells the hint-annotator that the value that arrives here is required to be a compile-time constant (i.e. green). In general, this is a very strong constraint, because it forces "backwards" a potentially large number of values to be green as well - all the values that this one depends on. In general, it does not work at all, because the value ultimately depends on an operation that cannot be constant-folded at all by the JIT compiler, e.g. because it depends on external input or reads from non-immutable memory.

The `promote=True` hint can take an arbitrary red value and returns it as a green variable, so it can be used to bound the set of values that need to be forced to green. A common idiom is to put a `concrete=True` hint at the precise point where a compile-time constant would be useful (e.g. on the value on which a complex switch dispatches), and then put a few `promote=True` hints to copy specific values into green variables *before* the `concrete=True`.

The `promote=True` hints should be applied where we expect not too many different values to arrive at runtime; here are typical examples:

- Where we expect a small integer, the integer can be promoted if each specialized version can be optimized (e.g. lists of known length can be optimized by the JIT compiler).
- The interpreter-level class of an object can be promoted before an indirect method call, if it is useful for the JIT compiler to look inside the called method. If the method call is indirect, the JIT compiler merely produces a similar indirect method call in the generated code. But if the class is a compile-time constant, then it knows which method is called, and compiles its operations (effectively inlining it from the point of the view of the generated code).
- Whole objects can be occasionally promoted, with care. For example, in an interpreter for a language which has function calls, it might be useful to know exactly which Function object is called (as opposed to just the fact that we call an object of class Function).

Other hints

The other hints mentioned in pypy/jit/tl/tiny2.py are "global merge points" and "deepfreeze". For more information, please refer to the explanations there.

We should also mention a technique not used in `tiny2.py`, which is the notion of *virtualizable* objects. In PyPy, the Python frame objects are virtualizable. Such objects assume that they will be mostly read and mutated by the JIT'ed code - this is typical of frame objects in most interpreters: they are either not visible at all for the interpreted programs, or (as in Python) you have to access them using some reflection API. The `_virtualizable_` hint allows the object to escape (e.g. in PyPy, the Python frame object is pushed on the globally-accessible frame stack) while still remaining efficient to access from JIT'ed code.

[JITSPEC] Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python, ACM SIGPLAN PEPM'04, August 24-26, 2004, Verona, Italy. http://psyco.sourceforge.net/psyco-pepm-a.ps.gz