Tracing the Meta-Level: PyPy’s Tracing JIT Compiler

Carl Friedrich Bolz  
University of Düsseldorf  
STUPS Group  
Germany  
cfbolz@gmx.de

Antonio Cuni  
University of Genova  
DISI  
Italy  
cuni@disi.unige.it

Maciej Fijalkowski  
merlinux GmbH  
fijal@merlinux.eu

Armin Rigo  
arigo@tunes.org

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

ABSTRACT

We attempt to apply the technique of Tracing JIT Compilers in the context of the PyPy project, i.e., to programs that are interpreters for some dynamic languages, including Python. Tracing JIT compilers can greatly speed up programs that spend most of their time in loops in which they take similar code paths. However, applying an unmodified tracing JIT to a program that is itself a bytecode interpreter results in very limited or no speedup. In this paper we show how to guide tracing JIT compilers to greatly improve the speed of bytecode interpreters. One crucial point is to unroll the bytecode dispatch loop, based on two kinds of hints provided by the implementer of the bytecode interpreter. We evaluate our technique by applying it to two PyPy interpreters: one is a small example, and the other one is the full Python interpreter.

1. INTRODUCTION

Dynamic languages have seen a steady rise in popularity in recent years. JavaScript is increasingly being used to implement full-scale applications which run within a browser, whereas other dynamic languages (such as Ruby, Perl, Python, PHP) are used for the server side of many web sites, as well as in areas unrelated to the web.

One of the often-cited drawbacks of dynamic languages is the performance penalties they impose. Typically they are slower than statically typed languages. Even though there has been a lot of research into improving the performance of dynamic languages (in the SELF project, to name just one example [20]), those techniques are not as widely used as one would expect. Many dynamic language implementa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
2. THE PYPY PROJECT

The PyPy project\footnote{http://codespeak.net/pypy} is an environment where flexible implementations of dynamic languages can be written. To implement a dynamic language with PyPy, an interpreter for that language has to be written in RPython\footnote{http://codespeak.net/pypy}. RPython ("Restricted Python") is a subset of Python chosen in such a way that type inference can be performed on it. The language interpreter can then be translated with the help of PyPy into various target environments, such as C/Posix, the CLI and the JVM. This is done by a component of PyPy called the translation toolchain.

By writing VMs in a high-level language, we keep the implementation of the language free of low-level details such as memory management strategy, threading model or object layout. These features are automatically added during the translation process. The process starts by performing control flow graph construction and type inferences, then followed by a series of steps, each step transforming the intermediate representation of the program produced by the previous one until we get the final executable. The first transformation step makes details of the Python object model explicit in the intermediate representation, later steps introducing garbage collection and other low-level details. As we will see later, this internal representation of the program is also used as an input for the tracing JIT.

3. TRACING JIT COMPILERS

Tracing optimizations were initially explored by the Dynamo project\footnote{http://codespeak.net/pypy} to dynamically optimize machine code at runtime. Its techniques were then successfully used to implement a JIT compiler for a Java VM\footnote{http://codespeak.net/pypy}. Subsequently these tracing JITs were discovered to be a relatively simple way to implement JIT compilers for dynamic languages\footnote{http://codespeak.net/pypy}. The technique is now being used by both Mozilla’s TraceMonkey JavaScript VM\footnote{http://codespeak.net/pypy} and has been tried for Adobe’s Tamarin ActionScript VM\footnote{http://codespeak.net/pypy}.

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

Typically tracing VMs go through various phases when they execute a program: Interpretation/profiling, tracing, code generation and execution of the generated code. When the program starts, everything is interpreted. The interpreter does lightweight profiling to establish which loops are run most frequently. This lightweight profiling is usually done by having a counter on each backward jump instruction that counts how often this particular backward jump is executed. Since loops need a backward jump somewhere, this method looks for loops in the user program.

When a hot loop is identified, the interpreter enters a special mode, called tracing mode. During tracing, 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 loop. 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. Such a trace can be used to generate efficient machine code. This generated machine code is immediately executable, and can be used in the next iteration of the loop. 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 direction, for example at conditional instructions or 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.\footnote{There are more complex mechanisms in place to still produce extra code for the cases of guard failures\footnote{http://codespeak.net/pypy}, but they are independent of the issues discussed in this paper.}

It is important to understand how the tracer recognizes that the trace it recorded so far corresponds to a loop. This happens when the position key is the same as at an earlier position. The position key describes the position of the program, i.e., usually contains things like the function currently being executed and the program counter position of the tracing interpreter. The tracing interpreter does not need to check all the time whether the position key already occurred earlier, but only at instructions that are able to change the position key to an earlier value, e.g., a backward branch instruction. Note that this is already the second place where backward branches are treated specially: during interpretation they are the place where the profiling is performed and where tracing is started or already existing assembler code executed; during tracing they are the place where the check for a closed loop is performed.

As a small example, take the (slightly contrived) RPython code in Figure 1. The tracer interprets these functions in a bytecode format that is an encoding of the intermediate rep-
representation of PyPy’s translation toolchain after type inference has been performed. When the profiler discovers that the while loop in strange_sum is executed often the tracing JIT will start to trace the execution of that loop. The trace would look as in the lower half of Figure 1.

The operations in this sequence are operations of the above-mentioned intermediate representation (e.g., the generic modulo and equality operations in the function above have been recognized to always take integers as arguments and are thus rendered as int_mod and int_eq). The trace contains all the operations that were executed in SSA-form [13] and ends with a jump to its beginning, forming an endless loop that can only be left via a guard failure. The call to f is inlined into the trace. The trace contains only the hot else case of the if test in f, while the other branch is implemented via a guard failure. This trace can then be converted into machine code and executed.

3.1 Applying a Tracing JIT to an Interpreter

The tracing JIT of the PyPy project is atypical in that it is not applied to the user program, but to the interpreter running the user program. In this section we will explore what problems this brings, and suggest how to solve them (at least partially). This means that there are two interpreters involved, and we need appropriate terminology to distinguish between them. On the one hand, there is the interpreter that runs the user’s programs, which we will call the language interpreter. In the following, we will assume that the language interpreter is bytecode-based. The program that the language interpreter executes we will call the user program (from the point of view of a VM author, the “user” is a programmer using the VM). Similarly, we need to distinguish loops at two different levels: interpreter loops are loops inside the language interpreter. On the other hand, user loops are loops in the user program.

A tracing JIT compiler finds the hot loops of the program it is compiling. In our case, this is the language interpreter. The most important hot interpreter loop is the bytecode dispatch loop (for many simple interpreters it is also the only hot loop). One iteration of this loop corresponds to the execution of one opcode. This means that the assumption made by the tracing JIT that several iterations of a hot loop take the same or similar code paths – is wrong in this case. It is very unlikely that the same particular opcode is executed many times in a row.

An example is given in Figure 2. It shows the code of a very simple bytecode interpreter with 256 registers and an accumulator. The bytecode argument is a string of bytes, all register and the accumulator are integers. A program for this interpreter that computes the square of the accumulator is shown in Figure 3. If the tracing interpreter traces the execution of the DECR_A opcode (whose integer value is 7), the trace would look as in Figure 4. 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.

To improve this situation, the tracing JIT could trace the

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
        if a:
            pc = target
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == ADD_R_TO_A:
            n = ord(bytecode[pc])
            pc += 1
            a += regs[n]
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a

MOV_A_R 0  # i = a
MOV_A_R 1  # copy of 'a'

# 4:
MOV_R_A 0  # i--
DECR_A
MOV_R_A 0

MOV_R_A 2  # res += a
ADD_R_TO_A 1
MOV_R_A 2

MOV_R_A 0  # if i!=0: goto 4
JUMP_IF_A 4
MOV_R_A 2  # return res
RETURN_A
```

Figure 2: A very simple bytecode interpreter with registers and an accumulator.

```
Figure 3: Example bytecode: Compute the square of the accumulator

loop_start(a0, regs0, bytecode0, pc0)
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(7))
ai = int_sub(a0, Const(1))
jump(ai, regs0, bytecode0, pc1)
```

Figure 4: Trace when executing the DECR_A opcode
execution of several opcodes, thus effectively unrolling the bytecode dispatch loop. Ideally, the bytecode dispatch loop should be unrolled exactly so much that the unrolled version corresponds to a user loop. User loops occur when the program counter of the language interpreter has the same value several times. This program counter is typically stored in one or several variables in the language interpreter, for example the bytecode object of the currently executed function of the user program and the position of the current bytecode within that. In the example above, the program counter is represented by the bytecode and pc variables.

Since the tracing JIT cannot know which parts of the language interpreter are the program counter, the author of the language interpreter needs to mark the relevant variables of the language interpreter with the help of 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 these variables that are making up the program counter at the language interpreter level are the same a second time. Loops found in this way are, by definition, user loops.

The program counter of the language interpreter can only be the same a second time after an instruction in the user program sets it to an earlier value. This happens only at backward jumps in the language interpreter. That means that the tracing interpreter needs to check for a closed loop only when it encounters a backward jump in the language interpreter. Again the tracing JIT cannot know which part of the language interpreter implements backward jumps, so the author of the language interpreter needs to indicate this with the help of a hint.

The language interpreter uses a similar technique to detect *hot user loops*: the profiling is done at the backward branches of the user program, using one counter per seen program counter of the language interpreter.

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.

Let’s look at how hints would need to be applied to the example interpreter from Figure 2. Figure 5 shows the relevant parts of the interpreter with hints applied. One needs to instantiate JitDriver by listing all the variables of the bytecode loop. The variables are classified into two groups, “green” variables and “red” variables. The green variables are those that the tracing JIT should consider to be part of the program counter of the language interpreter. In the case of the example, the pc variable is obviously part of the program counter; however, the bytecode variable is also counted as green, since the pc variable is meaningless without the knowledge of which bytecode string is currently being interpreted. All other variables are red.

```python
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        tlrjitdriver = JitDriver(greens = ['pc', 'bytecode'],
                        reds = ['a', 'regs'])
        pc += 1
        tlrjitdriver.jit_merge_point()
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                if target < pc:
                    tlrjitdriver.can_enter_jit()
                    pc = target
            elif opcode == MOV_A_R:
                # rest unmodified
```

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 bytecode dispatch loop. The other, more interesting one, is `can_enter_jit`. This method needs to be called at the end of any instruction that can set the program counter of the language interpreter to an earlier value.\(^4\) For the example this is only the `JUMP_IF_A` instruction, and only if it is actually 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. This is considered to be the case when the values of the “green” variables are the same as at an earlier call to the `can_enter_jit` method.

For the small example the hints look like a lot of work. However, the number of hints that need to be put into the interpreter source remains small, which makes the extra work negligible for larger interpreters.

When executing the Square function of Figure 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, thus unrolling the interpreter loop of the example interpreter eight times. The resulting trace can be seen in Figure 6.

### 3.2 Improving the Result

The critical problem of tracing the execution of just one opcode has been solved, the loop corresponds exactly to the loop in the square function. However, the resulting trace is not optimized enough. Most of its operations are not actually doing any computation that is part of the square function. Instead, they manipulate the data structures of the language interpreter. While this is to be expected, given that the tracing interpreter looks at the execution of the language interpreter, it would still be an improvement if some of these operations could be removed.

The simple insight on how to improve the situation is that most of the operations in the trace are actually concerned with manipulating the bytecode string and the pro-

\(^4\)The hints need to be written a bit differently in the actual implementation for purely technical reasons.
variable is

the check would be that at the beginning of the trace the
course of the loop, though). In the example of Figure 6
at the beginning of the loop (they may well change over the
tracer checks that those variables have some fixed value
(i.e., they are part of the position key). This means that
gram counter. Those are stored in variables that are “green”
and the
and a constant number, so they can be

Figure 6: Trace when executing the Square function of Figure 3, with the corresponding opcodes as comments.

by some other optimization, but is maybe not really all that
bad anyway (in fact we have an experimental optimization
that does exactly that, but it is not yet finished). Once we
get this optimized trace, we can pass it to the JIT backend,
which generates the corresponding machine code.

4. IMPLEMENTATION ISSUES

In this section we will describe some of the practical issues
when implementing the scheme described in the last section
in PyPy. In particular we will describe some of the problems
of integrating the various parts with each other.

The first integration problem is how to not integrate the
tracing JIT at all. It is possible to choose when the language
interpreter is translated to C whether the JIT should be built
in or not. If the JIT is not enabled, all the hints that are
possibly in the interpreter source are just ignored by the
translation process.

If the JIT is enabled, things are more interesting. At the
moment the JIT can only be enabled when translating the
interpreter to C, but we hope to lift that restriction in the
future. A classical tracing JIT will interpret the program it
is running until a hot loop is identified, at which point trac-
ing and ultimately assembler generation starts. The tracing
JIT in PyPy is operating on the language interpreter, which
is written in RPython. But RPython programs are statically
translatable to C anyway. This means that interpreting the
language interpreter before a hot loop is found is clearly not
desirable, since the overhead of this double-interpretation
would be significantly too big to be practical.

What is done instead is that the language interpreter
keeps running as a C program, until a hot loop in the user
program is found. To identify loops, the C version of the
language interpreter is the fact that the register list is still used
to store the state of the computation. This could be removed

Figure 7: Trace when executing the Square function of Figure 3, with the corresponding bytecodes as comments. The constant-folding of operations on green variables is enabled.

loop_start(a0, regs0, bytecode0, pc0)
# MOV_A_A 0
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(2))
n1 = strgetitem(bytecode0, pc1)
pc2 = int_add(pc1, Const(1))
a1 = list_getitem(regs0, n1)
# DECR_A
opcode1 = strgetitem(bytecode0, pc2)
ph

With this optimization enabled, the trace looks as in Fig-
ure 7. Now much of the language interpreter is actually gone
from the trace and what is left corresponds very closely to
the loop of the square function. The only vestige of the lan-

guard_true(i1)
jump(a5, regs0, bytecode0, target0)

i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)

loop_start(a0, regs0)
# MOV_A_A 0
a1 = list_getitem(regs0, Const(0))
# DECR_A
a2 = int_sub(a1, Const(1))
# MOV_A_R 2
list_setitem(regs0, Const(0), a2)
# MOV_A_R 2

a4 = int_add(a3, i0)
# MOV_R_A 2

n2 = strgetitem(bytecode0, pc4)
pc5 = int_add(pc4, Const(1))
list_setitem(regs0, n2, a2)
# MOV_A_R 2

n4 = strgetitem(bytecode0, pc8)
pc9 = int_add(pc8, Const(1))
i0 = list_getitem(regs0, n4)
# ADD_R_TO_A 1

n3 = strgetitem(bytecode0, pc13)
pc14 = int_add(pc13, Const(1))
guard_value(opcode6, Const(1))
target0 = strgetitem(bytecode0, pc14)
# MOV_A_R_A 0

a1 = list_getitem(regs0, Const(0))
# DECR_A
a2 = int_sub(a1, Const(1))
# MOV_A_R 2
list_setitem(regs0, Const(0), a2)
# MOV_A_R 2

i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)
tracing interpreter is invoked to start tracing the language interpreter that is running the user program. Of course the tracing interpreter cannot actually trace the execution of the C representation of the language interpreter. Instead it takes the state of the execution of the language interpreter and starts tracing using a bytecode representation of the language interpreter. That means there are two “versions” of the language interpreter embedded in the final executable of the VM: on the one hand it is there as executable machine code, on the other hand as bytecode for the tracing interpreter. It also means that tracing is costly as it incurs a double interpretation overhead.

From then on things proceed as described in Section 3. The tracing interpreter tries to find a loop in the user program, if it finds one it will produce machine code for that loop and this machine code will be immediately executed. The machine code is executed until a guard fails. Then the execution should fall back to normal interpretation by the language interpreter. This falling back is possibly a complex process, since the guard failure can have occurred arbitrarily deep in a helper function of the language interpreter, which would make it hard to rebuild the state of the language interpreter and let it run from that point (e.g., this would involve building a potentially deep C stack). Instead the falling back is achieved by a special fallback interpreter which runs the language interpreter and the user program from the point of the guard failure. The fallback interpreter is essentially a variant of the tracing interpreter that does not keep a trace. The fallback interpreter runs until execution reaches a safe point where it is easy to let the C version of the language interpreter resume its operation. This means that the fallback interpreter executes at most one bytecode operation of the language interpreter and then falls back to the C version of the language interpreter. After this, the whole process of profiling may start again.

Machine code production is done via a well-defined interface to an assembler backend. This allows easy porting of the tracing JIT to various architectures (including, we hope, to virtual machines such as the JVM where our backend could generate JVM bytecode at runtime). At the moment the only implemented backend is a 32-bit Intel-x86 backend.

5. EVALUATION

In this section we evaluate the work done so far by looking at some benchmarks. Since the work is not finished, these can only be preliminary. Benchmarking was done on an otherwise idle machine with a 1.4 GHz Pentium M processor and 1 GB RAM, using Linux 2.6.27. All benchmarks where run 50 times, each in a newly started process. The first run was ignored. The final numbers were reached by computing the average of all other runs, the confidence intervals were computed using a 95% confidence level. All times include the profile overhead, the confidence intervals were set so high that the tracer is never invoked. In this way the overhead of the profiling is measured. For this interpreter it seems to be rather large, with about 20% slowdown due to profiling. This is because the interpreter is small and the opcodes simple. For larger interpreters (e.g., PyPy’s Python interpreter) the overhead will likely be less significant.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Time (ms)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compiled to C, no JIT</td>
<td>442.7 ± 3.4</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>Normal Trace Compilation</td>
<td>1518.7 ± 7.2</td>
<td>0.29</td>
</tr>
<tr>
<td>3</td>
<td>Unrolling of Interp. Loop</td>
<td>137.6 ± 7.9</td>
<td>0.60</td>
</tr>
<tr>
<td>4</td>
<td>JIT, Full Optimizations</td>
<td>156.2 ± 3.8</td>
<td>2.83</td>
</tr>
<tr>
<td>5</td>
<td>Profile Overhead</td>
<td>515.0 ± 7.2</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Figure 8: Benchmark results of example interpreter computing the square of 10000000

BENCHMARK 2: The tracing JIT is enabled, but no interpreter-specific hints are applied. This corresponds to the trace in Figure 4. The threshold when to consider a loop to be hot is 40 iterations. As expected, this is not faster than the previous number. It is even quite a bit slower, probably due to the overheads, as well as non-optimal generated machine code.

BENCHMARK 3: The tracing JIT is enabled and hints as in Figure 5 are applied. This means that the interpreter loop is unrolled so that it corresponds to the loop in the square function. Constant folding of green variables is disabled, therefore the resulting machine code corresponds to the trace in Figure 6. This alone brings an improvement over the previous case, but is still slower than pure interpretation.

BENCHMARK 4: Same as before, but with constant folding enabled. This corresponds to the trace in Figure 7. This speeds up the square function considerably, making it nearly three times faster than the pure interpreter.

BENCHMARK 5: Same as before, but with the threshold set so high that the tracer is never invoked. In this way the overhead of profiling is measured. For this interpreter it seems to be rather large, with about 20% slowdown due to profiling. This is because the interpreter is small and the opcodes simple. For larger interpreters (e.g., PyPy’s Python interpreter) the overhead will likely be less significant.

```python
def f(a):
    i = 0
    while i < a:
        t = (t[1], t[2], t[0])
        i += t[0]
    return i
```

Figure 9: Benchmarked function and results for the Python interpreter running f(10000000)

To test the technique on a more realistic example, we did some preliminary benchmarks with PyPy’s Python interpreter. The function we benchmarked as well as the results can be seen in Figure 9. While the function may seem a bit constructed, executing it is still non-trivial, as a normal Python interpreter needs to dynamically dispatch nearly all of the involved operations, like indexing into the tuple, addition and comparison of i. We benchmarked PyPy’s Python interpreter with the JIT disabled, with the JIT enabled and

---

5 This is the only reason for the jit_merge_point hint.

6 The result will overflow, but for smaller numbers the running time is not long enough to sensibly measure it.
The results show that the tracing JIT speeds up the execution of this Python function significantly, even outperforming CPython. To achieve this, the tracer traces through the whole Python dispatching machinery, automatically inlining the relevant fast paths. However, the manually tuned Psyco still performs a lot better than our prototype (although it is interesting to note that Psyco improves the speed of CPython by only a factor of 3.29 in this example, while our tracing JIT improves PyPy by a factor of 6.54).

6. RELATED WORK

Applying a trace-based optimizer to an interpreter and adding hints to help the tracer produce better results has been tried before in the context of the DynamoRIO project [27], which has been a great inspiration for our work. They achieve the same unrolling of the interpreter loop so that the unrolled version corresponds to the loops in the user program. However the approach is greatly hindered by the fact that they trace on the machine code level and thus have no high-level information available about the interpreter. This makes it necessary to add quite a large number of hints, because at the assembler level it is not really visible anymore that e.g., a bytecode string is immutable. Also more advanced optimizations like allocation removal would not be possible with that approach.

There are quite a number of approaches that try to minimally enhance interpreters to generate code at runtime without actually writing a native compiler by hand. The goal of these is to get rid of dispatch overhead of typical interpreters while retaining ease of implementation. Pinnart and Riccardi [22] propose to copy fragments of the interpreter together for commonly occurring bytecode sequences to reduce dispatch overhead. However, dispatching is still needed to jump between such sequences and also when non-copyable bytecodes occur. Ertl and Gregg [14] go further and get rid of all dispatch overhead by stitching together the concatenated sequences by patching the copied machine code. Both techniques can speed up interpreters which large dispatch overhead a lot. However they will help less if the bytecodes themselves do a lot of work (as is the case with Python [7]) and the dispatch overhead is lower. On the other hand, our technique can do a better job by tracing inside the implementation of those bytecode and inlining common paths.

The standard approach for automatically producing a compiler for a programming language given an interpreter for it is that of partial evaluation [15, 21]. Conceptually there are some similarities to our work. In partial evaluation some arguments of the interpreter function are known (static) while the rest are unknown (dynamic). This separation of arguments is related to our separation of variables into those that should be part of the position key and the rest. In partial evaluation all parts of the interpreter that rely only on static arguments can be constant-folded so that only operations on the dynamic arguments remain.

Classical partial evaluation has failed to be useful for dynamic language for much the same reasons why ahead-of-time compilers cannot compile them to efficient code. If the partial evaluator knows only the program it simply does not have enough information to produce good code. Therefore some work has been done to do partial evaluation at runtime. One of the earliest works on runtime specialisation is Tempo for C [11, 10]. However, it is essentially a normal partial evaluator “packaged as a library”; decisions about what can be specialised and how, are pre-determined. Another work in this direction is DyC [19], another runtime specializer for C. Both of these projects have a problem similar to that of DynamoRIO. Targeting the C language makes higher-level specialisation difficult.

There have been some attempts to do dynamic partial evaluation, which is partial evaluation that defers partial evaluation completely to runtime to make partial evaluation more useful for dynamic languages. This concept was introduced by Sullivan [26] who implemented it for a small dynamic language based on lambda-calculus. It is also related to Psyco [23], a specializing JIT compiler for Python.

There is some work by one of the authors to implement a dynamic partial evaluator for Prolog [3]. There are also experiments within the PyPy project to use dynamic partial evaluation for automatically generating JIT compilers out of interpreters [25, 12]. So far those have not been as successful as we would like and it seems likely that they will be supplanted with the work on tracing JITs described here.

7. CONCLUSION AND NEXT STEPS

We have shown techniques for improving the results when applying a tracing JIT to an interpreter. Our first benchmarks indicate that these techniques work really well on small interpreters and first experiments with PyPy’s Python interpreter make it appear likely that they can be scaled up to realistic examples.

A lot of work remains. We are working on two main optimizations at the moment. Those are:

Allocation Removal: A key optimization for making the approach produce good code for more complex dynamic language is to perform escape analysis on the loop operation after tracing has been performed. In this way all objects that are allocated during the loop and do not actually escape the loop do not need to be allocated on the heap at all but can be exploded into their respective fields. This is very helpful for dynamic languages where primitive types are often boxed, as the repeated allocation of intermediate results is very costly.

Optimizing Frame Objects: One problem with the removal of allocations is that many dynamic languages are so reflective that they allow the introspection of the frame object that the interpreter uses to store local variables (e.g., SmallTalk, Python). This means that intermediate results always escape because they are stored into the frame object, rendering the allocation removal optimization ineffective. To remedy this problem we make it possible to update the frame object lazily only when it is actually accessed from outside of the code generated by the JIT.

Eventually we will need to apply the JIT to the various interpreters that are written in RPython to evaluate how widely applicable the described techniques are. Possible targets for such an evaluation would be the SPy-VM, a Smalltalk implementation [4]; a Prolog interpreter; PyGirl, a Gameboy emulator [6]; and also not immediately obvious ones, like Python’s regular expression engine.

If these experiments are successful we hope that we can reach a point where it becomes unnecessary to write a language specific JIT compiler and instead possible to just ap-
ply a couple of hints to the interpreter to get reasonably good performance with relatively little effort.

Acknowledgements
The authors would like to thank Michael Leuschel, Toon Verwaest and the referees of ICOOOLPS’09 for helpful comments on earlier versions of this paper.

8. REFERENCES
[12] A. Cuni, D. Ancona, and A. Rigo. Faster than C#: Efficient implementation of dynamic languages on .NET. Accepted at ICOOOLPS’09.