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

ABSTRACT

We attempt to use the technique of Tracing JIT Compilers in the context of the PyPy project, i.e., on 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 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 [18]), those techniques are not as widely used as one would expect. Many dynamic language implementations use completely straightforward bytecode-interpreters without any advanced implementation techniques like just-in-time compilation. There are a number of reasons for this. Most of them boil down to the inherent complexities of using compilation. Interpreters are simple to implement, understand, extend and port whereas writing a just-in-time compiler is an error-prone task that is made even harder by the dynamic features of a language.

A recent approach to getting better performance for dynamic languages is that of tracing JIT compilers [16, 7]. Writing a tracing JIT compiler is relatively simple. It can be added to an existing interpreter for a language, the interpreter takes over some of the functionality of the compiler and the machine code generation part can be simplified.

The PyPy project is trying to find approaches to generally ease the implementation of dynamic languages. It started as a Python implementation in Python, but has now extended its goals to be generally useful for implementing other dynamic languages as well. The general approach is to implement an interpreter for the language in a subset of Python. This subset is chosen in such a way that programs in it can be compiled into various target environments, such as C/Posix, the CLI or the JVM. The PyPy project is described in more detail in Section 2.

In this paper we discuss ongoing work in the PyPy project to improve the performance of interpreters written with the help of the PyPy toolchain. The approach is that of a tracing JIT compiler. Contrary to the tracing JITs for dynamic languages that currently exist, PyPy's tracing JIT operates "one level down", i.e., it traces the execution of the interpreter, as opposed to the execution of the user program. The fact that the program the tracing JIT compiles is in our case always an interpreter brings its own set of problems. We describe tracing JITs and their application to interpreters in Section 3. By this approach we hope to arrive at a JIT compiler that can be applied to a variety of dynamic languages, given an appropriate interpreter for each of them. The process is not completely automatic but needs a small number of hints from the interpreter author, to help the tracing JIT. The details of how the process integrates into the rest of PyPy will be explained in Section 4. This work is not finished, but has already produced some promising results, which we will discuss in Section 5.

The contributions of this paper are:

- Applying a tracing JIT compiler to an interpreter.
- Finding techniques for improving the generated code.

2. THE PYPY PROJECT

The PyPy project[1] [21, 5] is an environment where flex-

[1] http://codespeak.net/pypy

ible implementation of dynamic languages can be written. To implement a dynamic language with PyPy, an interpreter for that language has to be written in RPython [1]. 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 [2] to dynamically optimize machine code at runtime. Its techniques were then successfully used to implement a JIT compiler for a Java VM [16, 15]. Subsequently these tracing JITs were discovered to be a relatively simple way to implement JIT compilers for dynamic languages [7]. The technique is now being used by both Mozilla's TraceMonkey JavaScript VM [14] and has been tried for Adobe's Tamarin ActionScript VM [8].

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
- Execution of the generated code

At first, when the program starts, everything is interpreted. The interpreter does a small amount of 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 during tracing 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 sequential 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 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.[2]

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 point. The position key describes the position of the execution 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.

Let's look at 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 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 abovementioned 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 [12] 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.

[2]There are more complex mechanisms in place to still produce extra code for the cases of guard failures [15], but they are independent of the issues discussed in this paper.

```
def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b
def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result

# corresponding trace:
loop_header(result0, n0)
i0 = int_mod(n0, Const(46))
i1 = int_eq(i0, Const(41))
guard_false(i1)
result1 = int_add(result0, n0)
n1 = int_sub(n0, Const(1))
i2 = int_ge(n1, Const(0))
guard_true(i2)
jump(result1, n1)
```

Figure 1: A simple Python function and the recorded trace.

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 beween them. On the one hand, there is the interpreter that the tracing JIT uses to perform tracing. This we will call the *tracing interpreter*. On the other 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 program 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). Tracing one iteration of this loop means that the recorded trace corresponds to 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.[3] A program for

[3]The chain of `if`, `elif`, ... instructions checking the various opcodes is turned into a `switch` statement by one of PyPy's optimizations. Python does not have a `switch` statement.

```
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
            regs[n] = a
        elif opcode == MOV_R_A:
            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
```

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

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

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

    MOV_R_A     2   # res += a
    ADD_R_TO_A  1
    MOV_A_R     2

    MOV_R_A     0   # if i!=0: goto 4
    JUMP_IF_A   4

    MOV_R_A     2   # return res
    RETURN_A
```

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))
a1 = int_sub(a0, Const(1))
jump(a1, regs0, bytecode0, pc1)
```

Figure 4: Trace when executing the `DECR_A` opcode

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 probably not improved at all.

To improve this situation, the tracing JIT could trace the 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 already existing machine code also needs to be adapted to this new situation. In a classical tracing JIT there is at most 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

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

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

Figure 5: Simple bytecode interpreter with hints applied

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 (the fact that red variables need to be listed explicitly too is an implementation detail).

In addition to the classification of the variables, there are two methods of `JitDriver` that need to be called. Both of them receive as arguments the current values of the variables listed in the definition of the driver. The first one is `jit_merge_point` which needs to be put at the beginning of the body 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. 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 is essentially constant no matter how large the interpreter is, 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.

```
loop_start(a0, regs0, bytecode0, pc0)
# MOV_R_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 = call(Const(<* fn list_getitem>), regs0, n1)
# DECR_A
opcode1 = strgetitem(bytecode0, pc2)
pc3 = int_add(pc2, Const(1))
guard_value(opcode1, Const(7))
a2 = int_sub(a1, Const(1))
# MOV_A_R 0
opcode1 = strgetitem(bytecode0, pc3)
pc4 = int_add(pc3, Const(1))
guard_value(opcode1, Const(1))
n2 = strgetitem(bytecode0, pc4)
pc5 = int_add(pc4, Const(1))
call(Const(<* fn list_setitem>), regs0, n2, a2)
# MOV_R_A 2
opcode2 = strgetitem(bytecode0, pc5)
pc6 = int_add(pc5, Const(1))
guard_value(opcode2, Const(2))
n3 = strgetitem(bytecode0, pc6)
pc7 = int_add(pc6, Const(1))
a3 = call(Const(<* fn list_getitem>), regs0, n3)
# ADD_R_TO_A 1
opcode3 = strgetitem(bytecode0, pc7)
pc8 = int_add(pc7, Const(1))
guard_value(opcode3, Const(5))
n4 = strgetitem(bytecode0, pc8)
pc9 = int_add(pc8, Const(1))
i0 = call(Const(<* fn list_getitem>), regs0, n4)
a4 = int_add(a3, i0)
# MOV_A_R 2
opcode4 = strgetitem(bytecode0, pc9)
pc10 = int_add(pc9, Const(1))
guard_value(opcode4, Const(1))
n5 = strgetitem(bytecode0, pc10)
pc11 = int_add(pc10, Const(1))
call(Const(<* fn list_setitem>), regs0, n5, a4)
# MOV_R_A 0
opcode5 = strgetitem(bytecode0, pc11)
pc12 = int_add(pc11, Const(1))
guard_value(opcode5, Const(2))
n6 = strgetitem(bytecode0, pc12)
pc13 = int_add(pc12, Const(1))
a5 = call(Const(<* fn list_getitem>), regs0, n6)
# JUMP_IF_A 4
opcode6 = strgetitem(bytecode0, pc13)
pc14 = int_add(pc13, Const(1))
guard_value(opcode6, Const(3))
target0 = strgetitem(bytecode0, pc14)
pc15 = int_add(pc14, Const(1))
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)
```

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

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 program counter. Those are stored in variables that are "green" (i.e., they are part of the position key). This means that the tracer checks that those variables have some fixed value at the beginning of the loop (they may well change over the course of the loop, though). In the example of Figure 6 the check would be that at the beginning of the trace the `pc` variable is `4` and the `bytecode` variable is the bytecode string corresponding to the square function. Therefore it is possible to constant-fold computations on them away, as long as the operations are side-effect free. Since strings are immutable in RPython, it is possible to constant-fold the `strgetitem` operation. The `int_add` are additions of the green variable `pc` and a constant number, so they can be folded away as well.

With this optimization enabled, the trace looks as in Figure 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 language interpreter is the fact that the register list is still used to store the state of the computation. This could be removed 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 tracing 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

```
loop_start(a0, regs0)
# MOV_R_A 0
a1 = call(Const(<* fn list_getitem>), regs0, Const(0))
# DECR_A
a2 = int_sub(a1, Const(1))
# MOV_A_R 0
call(Const(<* fn list_setitem>), regs0, Const(0), a2)
# MOV_R_A 2
a3 = call(Const(<* fn list_getitem>), regs0, Const(2))
# ADD_R_TO_A  1
i0 = call(Const(<* fn list_getitem>), regs0, Const(1))
a4 = int_add(a3, i0)
# MOV_A_R 2
call(Const(<* fn list_setitem>), regs0, Const(2), a4)
# MOV_R_A 0
a5 = call(Const(<* fn list_getitem>), regs0, Const(0))
# JUMP_IF_A 4
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0)
```

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

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 generated in such a way that at the place that corresponds to the `can_enter_jit` hint profiling is performed using the program counter of the language interpreter. Apart from this bit of profiling, the language interpreter behaves in just the same way as without a JIT.

When a hot user loop is identified, tracing is started. The 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.[4] 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 running the tracer and producing machine code.

The first round of benchmarks (Figure 8) are timings of the example interpreter given in Figure 2 computing the square of 10000000 using the bytecode of Figure 3.[5] The results for various configurations are as follows:

Benchmark 1: The interpreter translated to C without including a JIT compiler.

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 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.

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

[4]This is the only reason for the `jit_merge_point` hint.

[5]The result will overflow, but for smaller numbers the running time is not long enough to sensibly measure it.

		Time (ms)	speedup
1	Compiled to C, no JIT	442.7 ± 3.4	1.00
2	Normal Trace Compilation	1518.7 ± 7.2	0.29
3	Unrolling of Interp. Loop	737.6 ± 7.9	0.60
4	JIT, Full Optimizations	156.2 ± 3.8	2.83
5	Profile Overhead	515.0 ± 7.2	0.86

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

```
def f(a):
    t = (1, 2, 3)
    i = 0
    while i < a:
        t = (t[1], t[2], t[0])
        i += t[0]
    return i
```

		Time (s)	speedup
1	PyPy compiled to C, no JIT	23.44 ± 0.07	1.00
2	PyPy comp'd to C, with JIT	3.58 ± 0.05	6.54
3	CPython 2.5.2	4.96 ± 0.05	4.73
4	CPython 2.5.2 + Psyco 1.6	1.51 ± 0.05	15.57

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

arbitrary, 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 CPython[6] 2.5.2 (the reference implementation of Python). In addition we benchmarked CPython using Psyco 1.6 [20], a specializing JIT compiler for Python.

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 [24], 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.

The standard approach for automatically producing a compiler for a programming language given an interpreter for it is that of partial evaluation [13, 19]. 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 [10, 9]. 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 [17], 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 [23] who implemented it for a small dynamic language based on lambda-calculus. It is also related to Psyco [20], 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 [22, 11]. 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

[6] `http://python.org`

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.

Furthermore both tracing and leaving machine code are very slow due to a double interpretation overhead and we might need techniques for improving those.

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 apply a couple of hints to the interpreter to get reasonably good performance with relatively little effort.

8. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. 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.

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

[3] C. F. Bolz. *Automatic JIT Compiler Generation with Runtime Partial Evaluation*. Master thesis, Heinrich-Heine-Universität Düsseldorf, 2008.

[4] C. F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. *Back to the Future in One Week — Implementing a Smalltalk VM in PyPy*, pages 123–139. 2008.

[5] C. F. Bolz and A. Rigo. How to *not* write a virtual machine. In *Proceedings of the 3rd Workshop on Dynamic Languages and Applications (DYLA 2007)*, 2007.

[6] C. Bruni and T. Verwaest. PyGirl: generating Whole-System VMs from High-Level prototypes using PyPy. In *Tools, accepted for publication*, 2009.

[7] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. 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.

[8] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. 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.

[9] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. *Dagstuhl Seminar on Partial Evaluation*, pages 54—72, 1996.

[10] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, United States, 1996. ACM.

[11] A. Cuni, D. Ancona, and A. Rigo. Faster than C#: Efficient implementation of dynamic languages on .NET. Submitted to *ICOOOLPS'09*.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

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

[14] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, M. Bebenita, M. Chang, M. Franz, E. Smith, R. Reitmaier, and M. Haghighat. Trace-based Just-in-Time type specialization for dynamic languages. In *PLDI*, 2009.

[15] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[16] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[17] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[18] U. Hölzle. Adaptive optimization for SELF: reconciling high performance with exploratory programming. Technical report, Stanford University, 1994.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.

[20] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 15–26, Verona, Italy, 2004. ACM.

[21] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.

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

[23] G. T. Sullivan. Dynamic partial evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 238–256. Springer-Verlag, 2001.

[24] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 50–57, San Diego, California, 2003. ACM.