Faster than C#: efficient implementation of dynamic languages on .NET*

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

Davide Ancona
DISI, University of Genova
Italy
davide@disi.unige.it

Armin Rigo
arigo@tunes.org

ABSTRACT

The Common Language Infrastructure (CLI) is a virtual machine expressly designed for implementing statically typed languages as C#, therefore programs written in dynamically typed languages are typically much slower than C# when executed on .NET.

Recent developments show that *Just In Time* (JIT) compilers can exploit runtime type information to generate quite efficient code. Unfortunately, writing a JIT compiler is far from being simple.

In this paper we report our positive experience with automatic generation of JIT compilers as supported by the PyPy infrastructure, by focusing on JIT compilation for .NET. Following this approach, we have in fact added a second layer of JIT compilation, by allowing dynamic generation of more efficient .NET bytecode, which in turn can be compiled to machine code by the .NET JIT compiler.

The main and novel contribution of this paper is to show that this *two-layers JIT* technique is effective, since programs written in dynamic languages can run on .NET as fast as (and in some cases even faster than) the equivalent C# programs.

The practicality of the approach is demonstrated by showing some promising experiments done with benchmarks written in a simple dynamic language.

1. INTRODUCTION

Implementing a dynamic language such as Python with a compiler rather than with an interpreter improves performances at the cost of an increasing complexity. Furthermore, generating code for high level virtual machines like CLI or JVM enhances portability and inter-operability.

Writing a compiler that targets the CLI or JVM is easier than targeting a real CPU, but it still requires a lot of work, as IronPython[1], Jython[2] and JRuby[3] demonstrate. Finally, if one really seeks for an efficent language implementation, *Just In Time* (JIT) compilation needs to be considered; only in this way the compiler can exploit runtime type information to generate quite efficient code. Note that JIT compilation has not to be confused with lazy compilation of IronPython and Jython which is exploited to save memory, since in these cases no runtime type information is ever used to generate more efficient code.

Unfortunately, writing a JIT compiler is a very complex task. To make this task simpler, the solution proposed by PyPy [11] is automatic generation of JIT compilers with the help of partial evaluation techniques: the user has only to provide an interpreter manually annotated with *hints* specifying how interpretation and JIT compilation has to be interleaved [2].

More precisely, in this paper we focus on the ability of generating a JIT compiler able to emit code for the .NET virtual machine. To our knowledge, this is the first experiment with an interpreter with two *layers* of JIT compilation, since, before being executed, the emitted code is eventually compiled again by .NET's own JIT compiler.

The main contribution of this paper is to demonstrate that the idea of *JIT layering* can give good results, as dynamic languages can be even faster than their static counterparts.

1.1 Overview of PyPy

The *PyPy* project[4] [12] was initially conceived to develop an implementation of Python which could be easily portable and extensible without renouncing efficiency. To achieve these aims, the PyPy implementation is based on a highly modular design which allows high-level aspects to be separated from lower-level implementation details. The abstract semantics of Python is defined by an interpreter written in a high-level language, called RPython [1], which is in fact a subset of Python where some dynamic features have been

*This work has been partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments and by the EU-funded project: IST 004779 PyPy (PyPy: Implementing Python in Python).

[1] http://www.codeplex.com/IronPython
[2] http://www.jython.org/
[3] http://jruby.codehaus.org/
[4] http://codespeak.net/pypy/

sacrificed to allow an efficient translation of the interpreter to low-level code[5].

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

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

Despite having been specifically developed for Python, the PyPy infrastructure can in fact be used for implementing other languages. Indeed, there were successful experiments of using PyPy to implement several other languages such as Smalltalk [4], JavaScript, Scheme and Prolog.

1.2 PyPy and JIT-Generation

One of the most important aspects that PyPy's translation toolchain can weave in is the *automatic generation of a JIT compiler*. This section will give a high-level overview of how the JIT-generation process works. More details can be found in [13] and [2].

The main difference between the JIT compilers generated by PyPy and the ones found in other projects like IronPython is that the latter compile code at the method granularity: they can do little to optimize most of the operations inside, because few assumptions can be made about the types of the arguments and the global state of the program. The PyPy JITs, on the other hand, work at a sub-method granularity, as described next.

When using PyPy, the first step is to write an interpreter for the chosen language. Since it must be fed to the translation toolchain, the interpreter has to be written in RPython. Then, to guide the process, we need to add few manual annotations (called hints) to the interpreter, in order to teach the JIT generator which information is important to know at compile-time. From these annotations, PyPy will statically generate an interpreter and a JIT compiler in a single executable (here a .NET executable).

The interesting property of the generated JIT compiler is to delay the compilation until it knows all the information needed to generate efficient code. In other words, at runtime, when the interpreter notice that it is useful to compile a given piece of code, it sends it to the JIT compiler; however, if at some point the JIT compiler does not know about something it needs, it generates a *callback* into itself and stops execution.

[5]But note that it's a full Python interpreter; RPython is only the language in which this interpreter is written.

Later, when the generated code is executed, the callback might be hit and the JIT compiler is restarted again. At this point, the JIT knows exactly the state of the program and can exploit all this extra knowledge to generate highly efficient code. Finally, the old code is patched and linked to the newly generated code, so that the next time the JIT compiler will not be invoked again. As a result, **runtime and compile-time are continuously interleaved**.

Potentially, the JIT compiler generates new code for each different run-time value seen in variables it is interested in. This implies that the generated code needs to contain some sort of updatable switch, called *flexswitch*, which can pick the right code path based on the run-time value. Typically, the value we switch on is the runtime dynamic type of a value, so that the JIT compiler has all information needed to produce very good code for that specific case.

Modifying the old code to link to the newly generated one is very challenging on .NET, as the framework does not offer any primitive to do this. Section 2 explains how it is possible to obtain this behaviour.

2. THE CLI JIT BACKEND

2.1 JIT layering

From the implementation point of view, the JIT generator is divided into a frontend and several backends. The goal of the frontend is to generate a JIT compiler which works as described in the previous sections. Internally, the JIT represents the compiled code as *flow graphs*, and the role of the backends is to translate flowgraphs into machine code.

At the moment of writing, three backends have been implemented: one for Intel *x86* processors, one for *PowerPC* processors, and one for the *CLI Virtual Machine*. The latter is special because instead of emitting code for a real CPU it emits code for a virtual machine[6]: before being executed, the generated code will be compiled again by the .NET JIT compiler.

Thus, when using the CLI backend, we actually have two JIT compilers at two different layers, each one specialized in different kinds of optimization. By operating at a higher level, our JIT can potentially do a better job in some contexts, as our benchmarks demonstrate (see Section 3). On the other hand, the lower-level .NET JIT is very good at producing machine code, much more than PyPy's own *x86* backend, for example. By combining the strengths of both we can get highly efficient machine code.

As usual, the drawback is that programs that runs for a very short period of time could run slower with JIT than without, due to the time spent doing the initial (double) compilation. Finally, it is important to underline that while we have directed our efforts to generate a JIT compiler able to emit very efficient code, the performance of the compiler itself has been neglected so far, but it is certainly an issue which will have to be considered in our future research.

2.2 Flexswitches

[6]By using the `Reflection.Emit` namespace and creating `DynamicMethodS`.

For a large part, implementing the CLI backend is easy and straightforward, as there is a close correspondence between most of the operations used by frontend's flowgraphs and the CLI instructions. Thus, we will not go into details for this part.

However the concept of *flexswitch*, as described in Section 1.2, does not have any direct equivalent in the CLI model, and it is hard to implement efficiently.

A flexswitch is a special kind of switch which can be dynamically extended with new cases. Intuitively, its behavior can be described well in terms of flow graphs: a flexswitch can be considered as a special flow graph block where links to newly created blocks are dynamically added whenever new cases are needed.

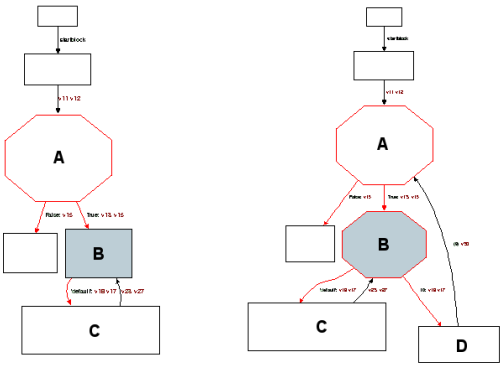

Figure 1: An example of a flexswitch evolution: in the picture on the right block D has been dynamically added.

In the pictures of Figure 1, block B (highlighted in grey) corresponds to a flexswitch; initially (picture on the left) only block C, containing the code to restart the JIT compilation, is connected to the flexswitch; the picture on the right shows the graph after the first case has been dynamically added to the flexswitch, by linking block B with the freshly created block D.

2.3 Implementing flexswitches in CLI

Implementing flexswitches for backends generating machine code is not too complex: basically, a new jump has to be inserted in the existing code to point to the newly generated code fragment.

Unfortunately, CLI does not allow modification of code which has been already loaded and linked, therefore the simplest approach taken for low level architectures does not work.

Since in .NET methods are the basic units of compilation, a possible solution consists in creating a new method any time a new case has to be added to a flexswitch.

It is important to underline the difference between flow graphs and a methods: the first are the logical unit of code as seen by the JIT compiler, each of them being concretely implemented by *one or more* methods.

In this way, whereas flow graphs without flexswitches are translated to a single method, the translation of *growable* flow graphs will be scattered over several methods. Summarizing, the backend behaves in the following way:

- Each flow graph is translated in a collection of methods which can grow dynamically. Each collection contains at least one method, called *primary*, which is the first to be created. All other methods, called *secondary*, are added dynamically whenever a new case is added to a flexswitch.
- Each either primary or secondary method implements a certain number of blocks, all belonging to the same flow graph.

When a new case is added to a flexswitch, the backend generates the new blocks into a new single method. The newly created method is pointed to by a delegate[7] stored in the flexswitch, so that it can be invoked later when needed.

2.3.1 Internal and external links

A link is called *internal* if it connects two blocks implemented by the same method, *external* otherwise.

Following an internal link is easy in IL bytecode: a jump to the corresponding code fragment in the same method can be emitted to execute the new block, whereas the appropriate local variables can be used for passing arguments.

Following an external link whose target is an initial block could also be easily implemented, by just invoking the corresponding method. What cannot be easily implemented in CLI is following an external link whose target is not an initial block; consider, for instance, the outgoing link from block D to block A in Figure 1. How is it possible to jump into the middle of a method?

To solve this problem every method contains a special code, called *dispatcher*: whenever a method is invoked, its dispatcher is executed first[8] to determine which block has to be executed. This is done by passing to the method a 32 bits number, called *block id*, which uniquely identifies the next block of the graph to be executed. The high 2 bytes of a block id constitute the *method id*, which univocally identifies a method in a graph, whereas the low 2 bytes constitute a progressive number univocally identifying a block inside each method.

The picture in Figure 2 shows a graph composed of three methods (for simplicity, dispatchers are not shown); method ids are in bold, whereas block numbers are in black. The graph contains three external links; in particular, note the link between blocks `0x00020001` and `0x00010001` which connects two blocks implemented by different methods.

The code[9] generated for the dispatcher of methods is similar

[7] *Delegates* are the .NET equivalent of function pointers.
[8] The dispatcher should not be confused with the initial block of a method.
[9] For simplicity we write C# code instead of the actual IL bytecode.

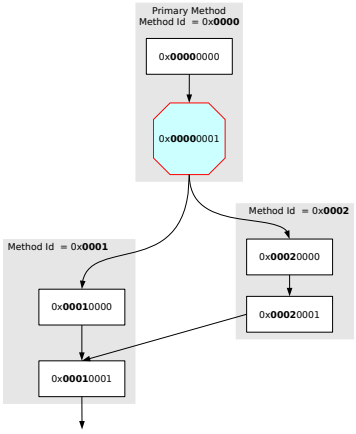

Figure 2: Method and block ids.

to the following fragment:

```
// dispatch block
int methodid = (blockid && 0xFFFF0000) >> 16;
int blocknum = blockid && 0x0000FFFF;
if (methodid != MY_METHOD_ID) {
  // jump_to_ext
  ...
}
switch(blocknum) {
  case 0: goto block0;
  case 1: goto block1;
  default: throw new Exception("Invalid block id");
}
```

If the next block to be executed is implemented in the same method (`methodid == MY_METHOD_ID`), then the appropriate jump to the corresponding code is executed. Otherwise, the `jump_to_ext` part of the dispatcher has to be executed, which is implemented differently by primary and secondary methods.

The primary method is responsible for the bookkeeping of the secondary methods which are added to the same graph dynamically. This can be simply implemented with an array mapping method id of secondary methods to the corresponding delegate. Therefore, the primary methods contain the following `jump_to_ext` code (where `FlexSwitchCase` is the type of delegates for secondary methods):

```
// jump_to_ext
FlexSwitchCase meth = method_map[methodid];
blockid = meth(blockid, ...); // execute the method
goto dispatch_block;
```

Each secondary method returns the block id of the next block to be executed; therefore, after the secondary method has returned, the dispatcher of the primary method will be executed again to jump to the correct next block.

To avoid mutual recursion and an undesired growth of the stack, the `jump_to_ext` code in dispatchers of secondary methods just returns the block id of the next block; since the primary method is always the first method of the graph which is called, the correct jump will be eventually executed by the dispatcher of the primary method.

Clearly this complex translation is performed only for flow graphs having at least one flexswitch; flow graphs without flexswitches are implemented in a more efficient and direct way by a unique method with no dispatcher.

2.3.2 *Passing arguments to external links*

The main drawback of our solution is that passing arguments across external links cannot be done efficiently by using the parameters of methods for the following reasons:

- In general, the number and type of arguments is different for every block in a graph;
- The number of blocks of a graph can grow dynamically, therefore it is not possible to compute in advance the union of the arguments of all blocks in a graph;
- Since external jumps are implemented with a delegate, all the secondary methods of a graph must have the same signature.

Therefore, the solution we came up with is defining a class `InputArgs` for passing sequences of arguments whose length and type is variable.

```
public class InputArgs {
  public int[] ints;
  public float[] floats;
  public object[] objs;
  ...
}
```

Unfortunately, with this solution passing arguments to external links becomes quite slow:

- When writing arguments, array re-allocation may be needed in case the number of arguments exceeds the dimension of the array. Furthermore the VM will always perform bound-checks, even when the size is explicitly checked in advance;
- When reading arguments, a bound-check is performed by the VM for accessing each argument; furthermore, an appropriate downcast must be inserted anytime an argument of type object is read.

Of course, we do not need to create a new object of class `InputArgs` any time we need to perform an external jump; instead, a unique object is created at the beginning of the execution of the primary method.

2.3.3 *Implementation of flexswitches*

To implement each flexswitch, the CLI backend creates an instace of a subclass of `BaseLowLevelFlexSwitch`: such an instance stores the mapping between each value and the corresponding method we want to invoke. Then, the generated code contains a call to the method `execute`, which selects and invoke the right method depending on the actual value we are switching on.

The following snippet shows the special case of integer flexswitches:

```
public class IntLowLevelFlexSwitch:
                        BaseLowLevelFlexSwitch {
  public uint default_blockid = 0xFFFFFFFF;
  public int numcases = 0;
  public int[] values = new int[4];
  public FlexSwitchCase[] cases =
                          new FlexSwitchCase[4];

  public void add_case(int value, FlexSwitchCase c) {
    ...
  }

  public uint execute(int value, InputArgs args) {
    for(int i=0; i<numcases; i++)
      if (values[i] == value)
        return cases[i](0, args);
    return default_blockid;
  }
}
```

The mapping from integers values to delegates (pointing to secondary methods) is just implemented by the two arrays `values` and `cases`. Method `add_case` extends the mapping whenever a new case is added to the flexswitch.

The most interesting part is the body of method `execute`, which takes a value and a set of input arguments to be passed across the link and jumps to the right block by performing a linear search in array `values`[10].

Recall that the first argument of delegate `FlexSwitchCase` is the block id to jump to. By construction, the target block of a flexswitch is always the first in a secondary method, and we use the special value `0` to signal this.

The value returned by method `execute` is the next block id to be executed; in case no association is found for `value`, `default_blockid` is returned. The value of `default_blockid` is initially set by the JIT compiler and usually corresponds to a block containing code to restart the JIT compiler for creating a new secondary method with the new code for the missing case, and updating the flexswitch by calling method `add_case`.

3. BENCHMARKS

To measure the performances of the CLI JIT backend, we wrote a simple virtual machine for a dynamic toy language, called *TLC*.

The design goal of the language is to be very simple (the interpreter of the full language consists of about 600 lines of RPython code) but to still have the typical properties of dynamic languages that make them hard to compile. TLC is implemented with a small interpreter that interprets a custom bytecode instruction set. Since our main interest is in the runtime performance of the interpreter, we did not implement the whole language, but just its virtual machine.

Despite being very simple and minimalistic, `TLC` is a good candidate as a language to test our JIT generator, as it has some of the properties that makes most of current dynamic languages (e.g. Python) so slow:

- **Stack based interpreter**: this kind of interpreter requires all the operands to be on top of the evaluation stack. As a consequence programs spend a lot of time pushing and popping values to/from the stack, or doing other stack related operations. However, thanks to its simplicity this is still the most common and preferred way to implement interpreters.
- **Boxed integers**: integer objects are internally represented as an instance of the `IntObj` class, whose field `value` contains the real value. By having boxed integers, common arithmetic operations are made very slow, because each time we want to load/store their value we need to go through an extra level of indirection. Moreover, in case of a complex expression, it is necessary to create many temporary objects to hold intermediate results.
- **Dynamic lookup**: attributes and methods are looked up at runtime, because there is no way to know in advance if and where an object has that particular attribute or method.

In the following sections, we present some benchmarks that show how our generated JIT can handle all these features very well.

To measure the speedup we get with the JIT, we run each program three times:

1. By plain interpretation, without any jitting (*Interp*).
2. With the JIT enabled: this run includes the time spent by doing the compilation itself, plus the time spent by running the produced code (*JIT*).
3. Again with the JIT enabled, but this time the compilation has already been done, so we are actually measuring how good is the code we produced (*JIT 2*).

Moreover, for each benchmark we also show the time taken by running the equivalent program written in C#.[11]

The benchmarks have been run on a machine with an Intel Pentium 4 CPU running at 3.20 GHz and 2 GB of RAM, running Microsoft Windows XP and Microsoft .NET Framework 2.0.

3.1 Arithmetic operations

To benchmark arithmetic operations between integers, we wrote a simple program that computes the factorial of a given number. The algorithm is straightforward, thus we are not showing the source code. The loop contains only three operations: one multiplication, one subtraction, and one comparison to check if we have finished the job.

When doing plain interpretation, we need to create and destroy three temporary objects (the results of each operation) at each iteration. By contrast, the code generated by the JIT

[10] Our microbenchmarks indicate that a linear search is the fastest way to find the right method to call, since typically each flexswitch contains only a very small number of cases.

[11] The sources for both TLC and C# programs are available at: `http://codespeak.net/ antocuni/tlc-benchmarks/`

does much better. At the first iteration, the classes of the two operands of the multiplication go through a flexswitch; then, the JIT compiler knows that both are integers, so it can inline the code to compute the result. Moreover, thanks to escape analysis, it can remove the allocation of all the temporary objects, because they never escape from the inner loop. The same remarks apply to the other two operations inside the loop.

As a result, the code executed after the first iteration is close to optimal: the intermediate values are stored as `int` local variables, and the multiplication, subtraction and *less-than* comparison are mapped to a single CLI opcode (`mul`, `sub` and `clt`, respectively).

Similarly, we wrote a program to calculate the n_{th} Fibonacci number, for which we can do the same reasoning as above.

Factorial

n	10	10^7	10^8	10^9
Interp	0.031	30.984	N/A	N/A
JIT	0.422	0.453	0.859	4.844
JIT 2	0.000	0.047	0.453	4.641
C#	0.000	0.031	0.359	3.438
Interp/JIT 2	N/A	**661.000**	N/A	N/A
JIT 2/C#	N/A	**1.500**	**1.261**	**1.350**

Fibonacci

n	10	10^7	10^8	10^9
Interp	0.031	29.359	0.000	0.000
JIT	0.453	0.469	0.688	2.953
JIT 2	0.000	0.016	0.250	2.500
C#	0.000	0.016	0.234	2.453
Interp/JIT 2	N/A	**1879.962**	N/A	N/A
JIT 2/C#	N/A	**0.999**	**1.067**	**1.019**

Table 1: Factorial and Fibonacci benchmarks

Table 1 shows the seconds spent to calculate the factorial and Fibonacci for various n. As we can see, for small values of n the time spent running the JIT compiler is much higher than the time spent to simply interpret the program. This is an expected result which, however, can be improved, since so far no effort has been direct to enhance the performance of the compiler itself.

On the other, for reasonably high values of n we obtain very good results, which are valid despite the obvious overflow, since the same operations are performed for all experiments. For n greater than 10^7, we did not run the interpreted program as it would have taken too much time, without adding anything to the discussion.

As we can see, the code generated by the JIT can be up to about 1800 times faster than the non-jitted case. Moreover, it often runs at the same speed as the equivalent program written in C#, being only 1.5 slower in the worst case.

The difference in speed it is probably due to both the fact that the current CLI backend emits slightly non-optimal code and that the underyling .NET JIT compiler is highly optimized to handle bytecode generated by C# compilers.

As we saw in Section 2.3, the implementation of flexswitches on top of CLI is hard and inefficient. However, our benchmarks show that this inefficiency does not affect the overall performances of the generated code. This is because in most programs the vast majority of the time is spent in the inner loop: the graphs are built in such a way that all the blocks that are part of the inner loop reside in the same method, so that all links inside are internal (and fast).

3.2 Object-oriented features

To measure how the JIT handles object-oriented features, we wrote a very simple benchmark that involves attribute lookups and polymorphic method calls. Since the TLC assembler source is long and hard to read, figure 3 shows the equivalent program written in an invented Python-like syntax.

```
def main(n):
    if n < 0:
        n = -n
        obj = new(value, accumulate=count)
    else:
        obj = new(value, accumulate=add)
    obj.value = 0
    while n > 0:
        n = n - 1
        obj.accumulate(n)
    return obj.value

def count(x):
    this.value = this.value + 1

def add(x):
    this.value = this.value + x
```

Figure 3: The *accumulator* example, written in a invented Python-like syntax

The two `new` operations create an object with exactly one field `value` and one method `accumulate`, whose implementation is found in the functions `count` and `add`, respectively. When calling a method, the receiver is implicity passed and can be accessed through the special name `this`.

The computation *per se* is trivial, as it calculates either $-n$ or $1+2...+n-1$, depending on the sign of n. The interesting part is the polymorphic call to `accumulate` inside the loop, because the interpreter has no way to know in advance which method to call (unless it does flow analysis, which could be feasible in this case but not in general). The equivalent C# code we wrote uses two classes and a `virtual` method call to implement this behaviour.

As already discussed, our generated JIT does not compile the whole function at once. Instead, it compiles and executes code chunk by chunk, waiting until it knows enough information to generate highly efficient code. In particular, at the time it emits the code for the inner loop it exactly knows the type of `obj`, thus it can remove the overhead of dynamic dispatch and inline the method call. Moreover, since `obj` never escapes the function, the allocation is avoided and its field `value` is stored as a local variable. As a result, the generated code turns out to be a simple loop doing additions in-place.

Accumulator				
n	10	10^7	10^8	10^9
Interp	0.031	43.063	N/A	N/A
JIT	0.453	0.516	0.875	4.188
JIT 2	0.000	0.047	0.453	3.672
C#	0.000	0.063	0.563	5.953
Interp/JIT 2	N/A	**918.765**	N/A	N/A
JIT 2/C#	N/A	**0.750**	**0.806**	**0.617**

Table 2: Accumulator benchmark

Table 2 show the results for the benchmark. Again, we can see that the speedup of the JIT over the interpreter is comparable to the other two benchmarks. However, the really interesting part is the comparison with the equivalent C# code, as the code generated by the JIT is up to 1.62 times **faster**.

Probably, the C# code is slower because:

- The object is still allocated on the heap, and thus there is an extra level of indirection to access the `value` field.
- The method call is optimized through a *polymorphic inline cache* [8], that requires a guard check at each iteration.

Despite being only a microbenchmark, this result is very important as it proves that our strategy of intermixing compile time and runtime can yield to better performances than current techniques. The result is even more impressive if we take in account that dynamically typed languages as TLC are usually considered much slower than the statically typed ones.

4. RELATED WORK

Flexswitches are closely related to the concept of *promotion*, as described by [13], [2]. Psyco is a run-time specialiser for Python that uses promotion (called "unlift" in [10]). However, Psyco is a manually written JIT, is not applicable to other languages and cannot be retargetted.

The idea of promotion is a generalization of *Polymorphic Inline Caches* [8], as well as the idea of using runtime feedback to produce more efficient code [9].

PyPy-style JIT compilers are hard to write manually, thus we chose to write a JIT generator. Tracing JIT compilers [7] also give good results but are much easier to write, making the need for an automatic generator less urgent. However so far tracing JITs have less general allocation removal techniques, which makes them get less speedup in a dynamic language with boxing. Another difference is that tracing JITs concentrate on loops, which makes them produce a lot less code. This issue is being addressed by current research in PyPy [3].

The code generated by tracing JITs code typically contains guards; in recent research [6] on Java, these guards' behaviour is extended to be similar to our promotion. This has been used twice to implement a dynamic language (JavaScript), by Tamarin[12] and in [5].

IronPython and Jython are two popular implementations of Python for, respectively, the CLI and the JVM, whose approach differs fundamentally from PyPy. The source code of PyPy contains a Python interpreter, which the JIT compiler is automatically generated from: the resulting executable contains both the interpreter and the compiler, so that it is possible to compile only the desired parts of the program. On the other hand, both IronPython and Jython implements only the compiler: both compile code lazily (when a Python module is loaded), but since they do not exploit the extra information potentially available at runtime, it is more a delayed static compilation than a true JIT one. As a result, they run Python programs much slower than their equivalent written in C#[13] or Java[14].

The *Dynamic Language Runtime*[15] (DLR) is a library designed to ease the implementation of dynamic languages for .NET: the DLR is closely related to IronPython[16] and employs the techniques described above; thus, the remarks about the differences between PyPy and IronPython apply to all DLR based languages.

5. CONCLUSION AND FUTURE WORK

In this paper we gave an overview of PyPy's JIT compiler generator, which can automatically turn an interpreter into a JIT compiler, requiring the language developers to only add few hints to guide the generation process.

Then, we presented the CLI backend for PyPy's JIT compiler generator, whose goal is to produce .NET bytecode at runtime. We showed how it is possible to circumvent intrinsic limitations of the virtual machine to implement flexswitches. As a result, we proved that the idea of *JIT layering* is worth of further exploration, as it makes possible for dynamically typed languages to be even faster than their statically typed counterpart in some cases.

As a future work, we want to explore different strategies to make the frontend producing less code, maintaining comparable or better performances. In particular, we are working on a way to automatically detect loops in the user code, as tracing JITs do [7]. By compiling whole loops at once, the backends should be able to produce better code.

At the moment, some bugs and minor missing features prevent the CLI JIT backend to handle more complex languages such as Python and Smalltalk. We are confident that once these problems will be fixed, we will get performance results comparable to TLC, as the other backends already demonstrate [13]. However, if the current implementation of flexswitches will turn out to be too slow for some purposes, alternative implementation strategies could be explored by

[12] `http://www.mozilla.org/projects/tamarin/`

[13] `http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=iron&lang2=csharp`

[14] `http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/`

[15] `http://www.codeplex.com/dlr`

[16] In fact, the DLR started as a spin-off of IronPython, and nowadays the latter is based on the former.

considering the novel features offered the new generation of virtual machines.

In particular, the *Da Vinci Machine Project* [17] is exploring and implementing new features to ease the implementation of dynamic languages on top of the JVM: some of these features, such as the new *invokedynamic*[18] instruction and the *tail call optimization* can probably be exploited by a potential JVM backend to generate even more efficient code.

6. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *OOPSLA 2007 Proceedings and Companion, DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64. ACM, 2007.

[2] D. Ancona, C. F. Bolz, A. Cuni, and A. Rigo. Automatic generation of JIT compilers for dynamic languages in .NET. Technical report, DISI, University of Genova and Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, 2008.

[3] C. F. Bolz, A. Cuni, A. Rigo, and M. Fijalkowski. Tracing the meta-level: Pypy's tracing jit compiler. Submitted to *ICOOOLPS'09*.

[4] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week - implementing a smalltalk vm in PyPy. In *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Revised Selected Papers*, volume 5146 of *Lecture Notes in Computer Science*, pages 123–139, 2008.

[5] 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, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

[6] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[7] 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.

[8] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

[9] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, New York, NY, USA, 1994. ACM.

[10] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, pages 15–26, 2004.

[11] A. Rigo and C. F. Bolz. How to not write Virtual Machines for Dynamic Languages . In *Proceeding of Dyla 2007 (to appear)*, pages –, 2007.

[12] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Companion*, pages 944–953, 2006.

[13] A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy Consortium, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[17] `http://openjdk.java.net/projects/mlvm/`
[18] `http://jcp.org/en/jsr/detail?id=292`