Making pypy happy

PyPy is a strange beast - it's possible to write incredibly fast code for PyPy and while we're trying to make it understand python code better, writing super fast code is still quite a bit of an arcane art. You can expect PyPy to outperform CPython by 2x with no effort, but you can reach 10x with effort that's very minimal compared to doing anything else with that impact. In this part I'll try to cover the basics of how to have a mindset that lets you write fast code. Maybe unsurprisingly, this has a lot more to do with "how to optimize Java" than "how to optimize Python for CPython". In fact most CPython-related optimizations are outright harmful.

CPyext vs cffi vs Python

The general rule is that using C extensions is slow. We've been doing quite a bit of work on that topic and we can expect to somehow converge to CPython speed over the next few months, but using CPython C extension API prevents us from doing any sensible optimizations, even if the costs of semantics mismatch of refcounting etc is negligable. cffi is the fastest way to call C in Python (it's really fast), but writing Python is often faster than calling CPython C extensions, because optimizations can be performed.

don't be afraid to call functions

Let's start with a few things you should not be afraid of that are well optimized in PyPy. Function calls, especially short functions that do very little and don't have loops are inlined and cheap or outright free

don't be afraid of objects

Same goes with objects and namedtuples and other stuff like that - objects are easy to do and very well optimized. Reading a known field from object should be a processor level memory dereference with an integer index.

pypy compiles everything*

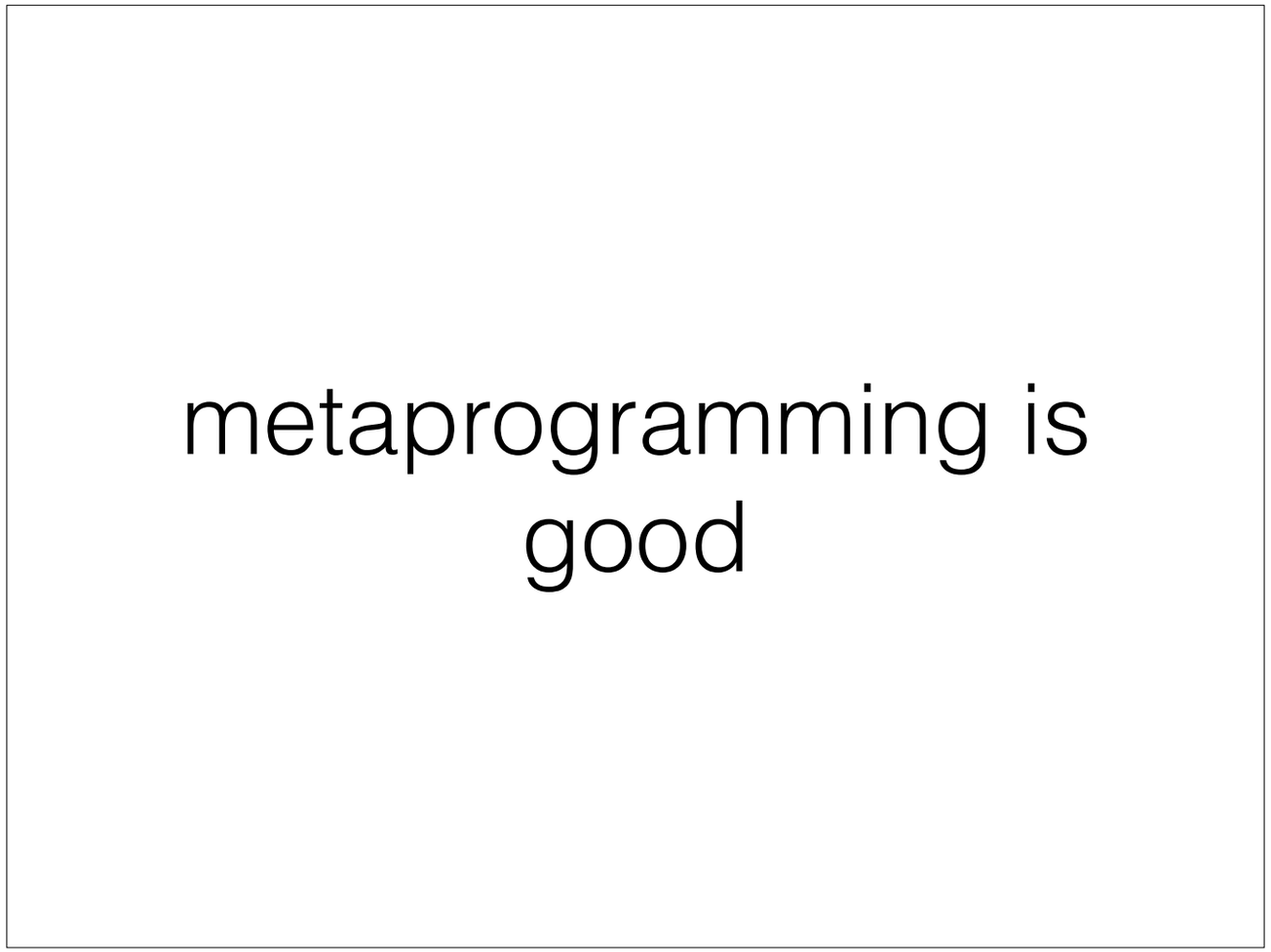
PyPy can compile everything by design, unlike in some JITs where the JIT compiler implements less of the language than the interpreter. The place where we disable the JIT on purpose is if you manipulate the frame objects all the time, since that creates traces that are just crazy.

avoid too general code

That said, too general code (aka Django ORM) is the worst possible case for PyPy. the general call after general call with parsing *args and **kwds, makes it impossible to have sane optimizations happen and you have to compile all the details. This makes speedups limited to about 2x in such cases as well as huge amount of assembler code produced and a lot of memory consumed.

avoid classes at runtime

Maybe obvious things, but the JIT compiler has tradeoffs that assume certain things like attribute access are far more common than things like class creation. Each class created has a lot of bookkeeping done in order to make other things done faster. This applies to a lot of dynamic code that's commonly slow in JITs - exec, functions at runtime etc.



That said, remember JIT does not read the source but follows the interpreter. It does not care that the code comes from exec or some other meta programming. This means that generating the code on the fly that's later simple is often a lot better than complicated code that gets dispatched at runtime. If I were to write an ORM for PyPy, I would compile per-SQL request or per callsite

itertools etc

Less known stuff: I would say try to avoid itertools and all the "goodies" from the standard library. There is no good reason why they should be slow, but python-dev keeps adding them more and more, usually implemented in C with strange semantics that are hard to get right, which makes it a lot harder for us to optimize and keep track of what's important what's not. There is also an issue of interactions between various layers of language that make it hard.

generators

We can do a much better job at generators. It's a researchy topic that requires some funding or a dedicated academic. They're not slow, but you'll end up having faster code not using them.

Use tools

This might be obvious, but it's crazy how many people have performance problems and never profile or write benchmarks

cProfile is inaccurate

One major thing - cProfile and all the tools that use cProfile can be widely inaccurate on PyPy. It can be inaccurate on CPython too, but the fact that you need to perform an operation per function call is a much more of a problem on optimizing compiler where eg abstract functions that do nothing can be really optimized away completely

vmprof

Our answer is vmprof - it creates relatively accurate data, but the visualization side of it could be improved. We would love to hire someone to do just tooling :)

reading traces

If you want to get absolutely max performance, you will have to write benchmarks including microbenchmarks, look at the traces created by those benchmarks and optimize them/see what's going on.

exercise

Ok, so here we go. I've written a simple serialization protocol for ints, objects and unicode strings. I've written two implementations - a slow one and a fast one and some benchmarks and API. Please try to start from a simple one and then try to optimize it. We can also use utf8 pypy and our temp_utf8 for extra speed