Dr Wenowdis: Specializing dynamic language C extensions using type information

Maxwell Bernstein
acm@bernsteinbear.com
Northeastern University
Boston, Massachusetts, USA

CF Bolz-Tereick
cfbolz@gmx.de
Heinrich-Heine-Universität
Düsseldorf, Germany

Abstract

C-based interpreters such as CPython make extensive use of C "extension" code, which is opaque to static analysis tools and faster runtimes with JIT compilers, such as PyPy. Not only are the extensions opaque, but the interface between the dynamic language types and the C types can introduce impedance. We hypothesise that frequent calls to C extension code introduce significant overhead that is often unnecessary.

We validate this hypothesis by introducing a simple technique, "typed methods", which allow selected C extension functions to have additional metadata attached to them in a backward-compatible way. This additional metadata makes it much easier for a JIT compiler (and as we show, even an interpreter!) to significantly reduce the call and return overhead.

Although we have prototyped typed methods in PyPy, we suspect that the same technique is applicable to a wider variety of language runtimes and that the information can also be consumed by static analysis tooling.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Runtime environments**.

ACM Reference Format:
Maxwell Bernstein and CF Bolz-Tereick. 2024. Dr Wenowdis: Specializing dynamic language C extensions using type information. In *Proceedings of SOAP '24.* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

One of the reasons for the success of dynamic languages such as Python and Ruby is the ease with which they can interface to existing C libraries through the use of C-implemented extension modules. Another common reason for writing C extensions is to improve the performance when the dynamic language runtime isn't fast enough in a hotspot. The downside of C extensions is that they cannot easily be analyzed by static analysis tools together with the Python or Ruby code that is calling into the library. The same problem plagues more advanced dynamic language implementations with a JIT compiler because a call into a C extension represents an optimization barrier [14]. For example, objects that may otherwise be unboxed by the JIT [5] now require boxing for consumption by the C extension, only to often be immediately unboxed again by the called C code.

As a motivating example, Listing 1 shows a complete minimal C extension module definition. The `inc` function takes and returns a Python `int` object. It unboxes its argument, increments it, and re-boxes the result. This type information is not available to the Python runtime; it's implicit in the C argument processing wrapper code. Calling this function from an optimizing Python implementation such as PyPy is thus very costly since it requires a generic call path and the allocation of C data structures that behave like the C extension expects them to.

In this paper we propose Dr Wenowdis [28], a very lightweight mechanism to expose some amount of type and effect knowledge about the functions a C extension module implements. We carry out this work specifically in the context of the CPython C API. We want to make it possible to incrementally add this knowledge to existing libraries without having to do an invasive rewrite or introduce a new dependency. We use the exposed information for improving the performance of Python→C calls using the PyPy JIT compiler [6]. The same type information can also be used for type checking (e.g. in MyPy [24]) or static analysis.

For the example, we can add the type information that the function takes and returns a C `long` by writing the code in Listing 2. This annotation is enough to speed up calling the `inc` function in PyPy by about 60 times, because it can call the `inc_impl` function directly, and optimize away the argument checking and unboxing. The example will be discussed more thoroughly in Section 2.2 and Section 3.

We present an early prototype that requires manual annotations with very limited expressiveness but a more complete version could generate the annotations automatically using a binding generator such as Cython [4] or PyO3 [17].

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

2 Background

2.1 CPython

CPython is the reference implementation of Python. Older versions implement the Python language by compiling it into a simple stack-based bytecode and running that in a straightforward interpreter [3, 31]. More recent versions of CPython (from 3.11 onwards) use bytecode quickening [8–11] to speed up bytecode execution [19]. The upcoming 3.13 release is probably also going to contain a simple baseline JIT [12] based on the copy-and-patch approach [35]. CPython uses reference counting in combination with a cycle-finding garbage collector to manage its memory. CPython boxes all of its objects, including integers and floating point numbers. It does not use pointer tagging or similar techniques.

2.2 The CPython C API

While writing Python code is the normal way of interacting with the CPython runtime, it is also possible to interact with it using its C API. The C API is commonly used to create C extension modules, which expose new functions and data types to Python code, that are implemented in C.[1] The C API consists of a number of free functions and data types, some of which are opaque to the API client. It gives the tools to create Python objects, introspect them, call Python functions, and more from C.

As an example, a minimal C extension can be found in Listing 1. In this C extension, `PyInit_signature` sets up the module. It calls the C API function `PyModule_Create`, which takes a description of the module it wants to create: the `PyModuleDef`. In the struct, we only define the minimal features for this example: a name, documentation, and a method table.

`PyModule_Create` walks the method table (the array of `PyMethodDef`), creating `PyCFunctionObjects` from the descriptions. In this example, it creates a C function called `inc` that takes one argument (indicated by the flag `METH_O`). When called, the C extension wrapper code inside the Python runtime does argument count checking and then passes the C function `inc` the singular argument it needs.

Extension authors are typically required to write their own argument processing code.[2] In this case, we convert the argument from a Python `int` object to a C `long` and raise an exception if that is not possible (this happens in `PyLong_AsLong`). Then we call the underlying C function, `inc_impl`, and box up the result for consumption in Python.

[1]The C API also makes it possible to embed CPython into other projects, but this usecase is much less frequent and we won't discuss it in this paper.
[2]There is a CPython-internal preprocessor called Argument Clinic that automates some of the work in writing argument processing code, but it is not meant for external projects. We discuss it in Section 6.2.

```c
#include <Python.h>

long inc_impl(long arg) { return arg+1; }

PyObject* inc(PyObject* module, PyObject* obj) {
  long l = PyLong_AsLong(obj);
  if (l == -1 && PyErr_Occurred()) return NULL;
  return PyLong_FromLong(inc_impl(l));
}

static PyMethodDef signature_methods[] = {
  {"inc", inc, METH_O, "Add one to a long."},
  {NULL, NULL, 0, NULL},
};

static struct PyModuleDef def = {
  PyModuleDef_HEAD_INIT, "signature", "doc", -1,
    signature_methods, NULL, NULL, NULL, NULL };

PyMODINIT_FUNC PyInit_signature(void) {
  return PyModule_Create(&def);
}
```

Listing 1. A tiny C extension, `signature`, exposing one function callable from Python, `inc`. The function `PyInit_signature` is called on first import.

```c
SIG(inc, LIST(T_C_LONG), T_C_LONG)
static PyMethodDef signature_methods[] = {
  TYPED_SIG(inc, inc, METH_O, "doc"),
  {NULL, NULL, 0, NULL},
};
```

Listing 2. Adding typing information to the minimal C extension in Listing 1.

2.3 PyPy

PyPy is an alternative implementation of the Python language. PyPy is not implemented in C, but in RPython, a statically typed subset of Python 2 [2]. PyPy uses a moving generational garbage collector for managing its memory. PyPy contains a tracing just-in-time compiler to speed up the execution of Python code [6]. To help the JIT compiler generate better code, PyPy's object model is quite different than that of CPython. In particular, Python instances are implemented using Self-style [13] maps/hidden classes [7].

2.4 cpyext and its problems

To allow PyPy to use the vast quantity of C extensions that exist for CPython, PyPy has a compatibility layer for the CPython C API, called *cpyext* [15]. It exposes (a subset of) the functions and structs of the C API.

Implementing this compatibility layer is quite challenging because CPython and PyPy function quite differently. The CPython C API exposes a number of internal implementation

details of CPython, most noticeably CPython's choice of reference counting for memory management. Handling Python objects from C requires the correct usage of `Py_INCREF` and `Py_DECREF` everywhere in the C code.[3] PyPy objects don't have a reference count field as the first word in each object, and the PyPy GC would really like to be able to move objects as part of its minor collections. Therefore, PyPy creates tiny CPython-layout compatible structs for those of its objects that that are passed to C functions.

Maintaining the link and converting between PyPy objects and CPython-layout compatible PyObjects is expensive. Every time PyPy calls a C function, we need to convert all of the function arguments to PyObjects and then convert the result back to a PyPy object. This is particularly expensive for boxed primitive types, because the C code will very likely just unbox them (with API functions such `PyLong_AsLong`) to work with the primitive values. However, because all the argument parsing and unboxing is done in C code and is therefore a black box, PyPy has no way to circumvent it. This is the central problem that we want to address in this paper and is visualized in Figure 1.

The problem becomes even more pronounced when PyPy's JIT is involved. The JIT will often compile the Python code that calls a C-implemented function in a C extension module. The JIT infers the types of variables that are used in Python code at run-time. In the case of integers the JIT will unbox them and store their integer values in machine registers [5]. In order to now pass such an unboxed integer as an argument to a C function, the JIT first has to recreate (i.e. allocate) a PyPy object, which is then converted to a PyObject in order to pass it to the C code. The C code will then unbox the value to work with the integer value itself. This kind of ping-pong between various representations is incredibly costly.

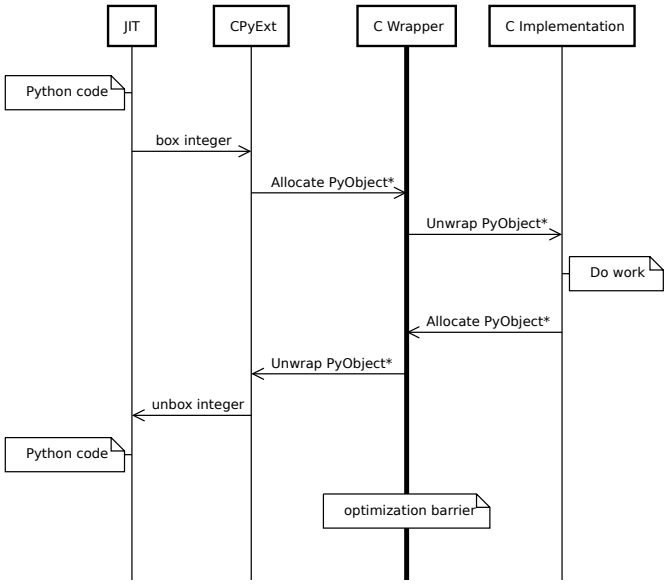

Figure 1. Example call to from Python code optimized by the JIT to a C function, passing one integer argument. The diagram shows all the needed conversions in the process.

3 Adding type information to C extensions

We want to pass known type information from the C extension functions back to Python. Doing this in a backwards compatible manner is non-trivial. We describe some of the problems of doing so in this section.

3.1 Exposing Type Information

Our broad goal is to allow the runtime—the *caller*—to make decisions about argument type checking and unboxing instead of the C extension—the callee. To do that, the runtime needs to know some type information and other metadata about each C function.

In order to make this work, adding our type information must be backwards-compatible both with PyPy and CPython. By this we mean that any version of CPython or PyPy that does not understand the annotations should not be tripped up by them: the C extensions should compile, load, and run just fine. This is tricky because the Python C API is not very extensible and also because of some guarantees that Python makes about its C API.

3.2 The stable ABI

The C API exposed by CPython is consumed not only be extension authors, but also by CPython developers. Over the years, API clients have come to rely on details that have been exposed by the CPython C API, such as struct sizes and structs fields that were originally intended for internal use only. Some source code has been completely lost to time, and all people have are shared objects that call into C API code. To support this CPython has defined a stable binary interface (ABI).

Under the stable ABI contract, functions are not removed, functions do not add or remove parameters, and data types do not change size. There are a few additional minor restrictions.

If we're going to try and make existing C API interactions faster under PyPy with minimal effort, we need to find a way to add lightweight annotations to methods. We can't change types, we can't change functions, and we can't make people work too hard.

Because both the size of `PyMethodDef`[4] and the sizes and types of its fields cannot change, we must smuggle in a pointer to more information stored elsewhere.

3.3 Sneaking in pointers

We can at least signal that there *is* additional information for a `PyMethodDef` by taking another bit in the `ml_flags` bitset.

[3] This includes the implicit runtime-owned wrapper code around C extension function calls.

[4] https://docs.python.org/3/c-api/structures.html#c.PyMethodDef

We propose the `METH_TYPED` bit. When this bit is set, the PyPy extension module loader knows to look for the extra type information.

```
struct PyMethodDef {
    const char  *ml_name;
    void        *ml_meth;
    int         ml_flags;
    const char  *ml_doc;
};
typedef struct PyMethodDef PyMethodDef;
```

Instead of the usual C string literal assigned to `ml_name`, we store the string in the `PyPyTypedMethodMetadata` struct and point `ml_name` to that buffer. The name size we chose is arbitrary. We then calculate an offset from that field to the beginning of the struct to use the added fields.

```
struct PyPyTypedMethodMetadata {
  int* arg_types;  // sentinel value of -1
  int ret_type;    // negative => can raise
  void* underlying_func;
  const char ml_name[100];
};
typedef struct PyPyTypedMethodMetadata
    PyPyTypedMethodMetadata;
```

A sample typed method looks like:

```
int inc_arg_types[] = {T_C_LONG, -1};
struct PyPyTypedMethodMetadata inc_sig = {
  inc_arg_types, T_C_LONG, inc_impl, "inc",
};
static PyMethodDef signature_methods[] = {
  {inc_sig.ml_name, inc, METH_O | METH_TYPED, "doc"
    },
  {NULL, NULL, 0, NULL},
};
```

To make this less irritating to write, we also provide macros to reach the form that we already saw in Listing 2. The macros also provide another feature: backwards compatibility. Instead of doing `#ifdef` yourself for type signature feature detection, the macros do it for you. On runtimes that support the `METH_TYPED` flag, they emit signatures. On runtimes that do not, they emit only standard C API method metadata.

Once we know that the type information exists, we can use a trick from the Linux kernel [22] and read backwards from the `ml_name`:

```
PyPyTypedMethodMetadata*
GetTypedSignature(PyMethodDef* def)
{
  return (PyPyTypedMethodMetadata*)(def->ml_name -
      offsetof(PyPyTypedMethodMetadata, ml_name));
}
```

4 Using type information in PyPy

Once the argument and return type information is in place for a C extension, this information can be used in cpyext. When we load a C extension module into PyPy, we load the module's methods. We check if each method has a `METH_TYPED` flag set. If it does, we find the metadata, build the signature, and store it on the internal method object.

When the function is called from PyPy, we first check whether the called function has type information attached. If that is the case, cpyext can use a fast path for implementing the call. The arguments that are declared to be primitive types can be type-checked on the PyPy side, without reboxing and subsequent conversion to `PyObject*`. The call can then use the `underlying_func` function pointer and therefore skip the overhead of whatever Python calling convention the function uses.

Being able to do the type checks for primitive arguments on the PyPy side (as opposed to doing it in C) also meshes with PyPy's JIT type annotation, which means the type check may not be required at all.

Last, instead of doing the slow and generic exception check, PyPy knows if the function may never raise an exception—so it need not check—or what special sentinel value to look for if it does raise. Functions can return `NULL`, or `-1`, or something else depending on the return type to signal an error. This fast value check removes the need for the full `PyErr_Occurred()` call in the case where the function did not signal that it raised. It is similar to CPython's existing strategy for exception checking.

5 Evaluation

To evaluate our changes, we compare our modified PyPy against mainline PyPy, CPython[5] [6], and GraalPy. We also measure our modified PyPy *with the JIT disabled* against mainline PyPy with the JIT disabled.

At this early stage of our research we are only using some micro-benchmarks. Every micro-benchmark is calling a C function many times in a hot loop. The different benchmarks exercise different kinds of calls from Python into native code. All the C function are themselves doing very little actual work. This means that the performance is dominated by the overheads of the C API and converting between the different representations. The results therefore represent the performance ceiling: the best possible improvements our approach can can make. They are not meant to represent real-world code. The four microbenchmarks are:

- `ffibench`, calling a `METH_O` function with C types `long→long`

[5] We also tested the alpha release of the upcoming CPython 3.13 and it gave similar, if slightly slower results than CPython 3.12.

[6] We would have also liked to benchmark against the Cinder JIT, but the open-source build for the JIT was broken at the time of writing.

- `objbench`, calling a `METH_FASTCALL` function with C types `PyObject* → long → long`
- `idbench`, calling a `METH_O` identity function with C types `PyObject* → PyObject*`.
- `idbench_exc`, also calling a `METH_O` identity function with C types `PyObject* → PyObject*`, but this variant is annotated with the information that it can raise an exception.

We run each benchmark 3 times for 1 billion iterations.[7] We then make a box and whisker plot of time taken for the three runs. There is not much variance per runtime per benchmark. The results can be seen in Figure 2.

The best case benchmark for optimization is `ffibench`. While the wrapper function deals in heap allocated Python `int`s, the underlying C function takes and returns a `long`. The type information lets PyPy's JIT skip the overhead of creating `PyObject*` for the argument and the return value completely. Our changes bring PyPy from slowest (about 164 seconds) to fastest (about 2.7 seconds).

In `objbench` one of the C-functions arguments is unboxed and the other one requires allocating a `PyObject` for each call. We do this to approximate a more realistic call. Not all calls to C extensions are going to be able to avoid all boxing. In this benchmark, even though PyPy is still allocating a `PyObject`, removing the overhead from building an array of `PyObject` for the `METH_FASTCALL` convention and from boxing the result makes a big difference. Our changes bring PyPy 3.10 from second slowest to fastest.

The third benchmark, `idbench`, benchmarks the identity function. It is a `METH_O` function (one parameter) but this time it cannot type-specialize its parameters or return value. This means that the runtime must box up the argument into a `PyObject` and unbox the return value the same way. Unlike with `METH_FASTCALL`, we are not eliminating any overhead for allocating an argument tuple/array. Despite this, we cut PyPy's execution time in half and are a close second place for time. GraalPy 22 is fastest, we suspect due to Sulong being able to optimize the call to the identity function.

The last benchmark, `idbench_exc`, is similar to `idbench` except that it can raise an exception. This means that the runtime must check first for an agreed-upon sentinel value (in this case, `NULL`) and second check for an exception. Even though we are checking exceptions, we cut PyPy's execution time in half and are a close second place for time.

Last, we ran all four benchmarks in PyPy with the JIT turned off (Figure 3). Our changes still improve PyPy performance because the runtime need not allocate the arguments array of `PyObjects` for each call to a `METH_FASTCALL` function; it knows how many arguments to pass and can pass them in registers when calling the underlying function. Additionally, the runtime can skip slow thread-local storage (TLS) lookups in exception checking for C extension functions that cannot raise Python exceptions.

[7] We specifically picked such high iteration counts to to give the Graal JIT enough time to warm up and have Sulong (Section 6.1) kick in for GraalPy 22. We omit GraalPy 23 because it took much longer than the other runtimes to finish and we killed the process. We hypothesize this is due to the removal of Sulong between versions 22 and 23. This makes GraalPy 23+ another good candidate for using type information from C extensions.

6 Prior work

Monat et al. [30] built a multi-language static analysis platform called Mopsa. They analyze several open-source libraries and their unit tests and find that multiple Python↔C calls happen per unit test (ranging from 2.7 to 51.7). Their analysis could potentially be repurposed to generate type information for C extensions (they note that they would like to infer standard library type information using their techniques). They note that the analysis is limited by use of hard-to-analyze Python libraries and imprecision in the C analysis. This work is expanded in Monat [29].

Similarly, Hu et al. [21] describe PyCType, which automatically infers the argument types for functions exposed by Python C extension in order to find bugs. This information should be usable for runtime performance improvements.

Tsai et al. [33] use the LLVM JIT to speed up the performance of JNA callbacks in the Java Hotspot Server VM [33]. Their approach yields 8-16% performance improvements and does not apply to *calling* C functions from Java (only the other way around).

Li and Tan [25, 26] find bugs in Java Native Interface (JNI) modules related to exception-checking. Their tools, JET and TurboJet, implement a static analysis to find missing declarations for checked exceptions. Automatically finding and annotating C extensions that do not raise exceptions could help improve run-time performance with less manual work.

The HPy project [1] is a complete re-design of the CPython C API from the ground up. One of its main goals is to move away from reference counting being visible in the API. However, it still does nothing to solve the problems discussed in this paper.

6.1 Sulong

Sulong [32] is an self-optimizing interpreter based on the Truffle framework [34] for LLVM [23] bitcode. It can be used to speed up calling from Python into C extensions by JIT-compiling both the Python and the C code in the same compilation unit using the Graal JIT compiler. This gets rid of most of the conversion overhead, but has the downside that called C function is also running on top of Sulong, which is slower than using a well-tuned static C compiler for the core algorithms of extensions.

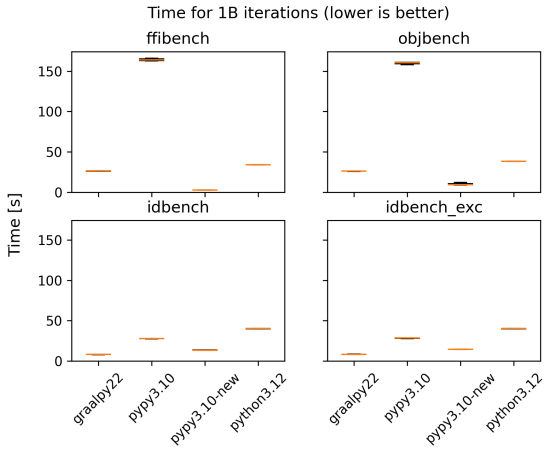

Figure 2. Benchmark results

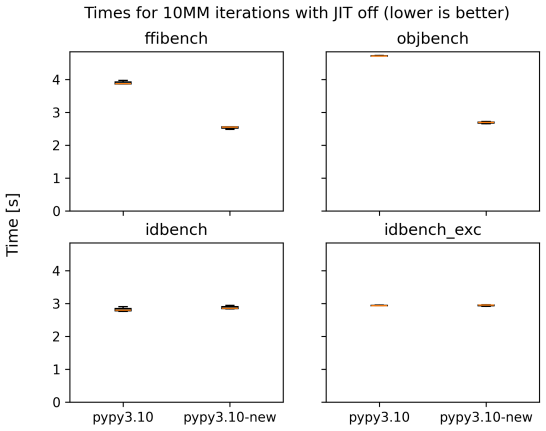

Figure 3. Benchmark for PyPy with JIT turned off

6.2 Argument Clinic

The CPython runtime has an internal tool called the Argument Clinic [20] that takes as input descriptions of C functions and generates argument processing code and documentation strings for them. The clinic is used only to generate standard library code in CPython.

6.3 Static Python

The Cinder project[8] is perhaps most widely known for its JIT compiler, but it also includes a compiler and runtime for a statically-typed dialect of Python known as Static Python (SP) [27]. The SP compiler contains primitives for declaring, resolving, and calling C functions directly from Static Python code. The SP compiler produces typed bytecode, which allows the JIT to compile specialized, zero-overhead calls to these C functions. See Listing 3 for an example.

```
1 from __static__ import native, int32, box
2
3 @native("libc.so.6")
4 def abs(i: int32) -> int32:
5     pass
6
7 def abs_wrapper(i: int) -> int:
8     j: int32 = int32(i)
9     return box(abs(j))
```

Listing 3. Taken from the Cinder test suite. In this example, the `abs` function is declared as a stub to be loaded from `libc.so`. It takes an unboxed (C) `int32` and returns the same. The `test` function takes a boxed (Python) `int`, unboxes, calls `abs`, and re-boxes.

The JIT uses `dlsym` and the typed SP bytecode to build a `NativeTarget`: a function pointer, return type, and argument types.

[8] https://github.com/facebookincubator/cinder/

Annotating declaration of existing C functions is a manual process. Also, the SP compiler does not allow passing `int`s into the unboxed `abs` function; callers must explicitly unbox.

7 Conclusion

We have shown that adding type information to C extensions can make them faster under the PyPy JIT. We have also shown that the techniques improve performance on PyPy even with the JIT compiler turned *off*.

Type information specialization is effective even in an interpreted context and potentially even without unboxed objects. We believe that this technique is not limited to PyPy and can be adopted by other dynamic language runtimes. For example, in runtimes such as Skybison and MicroPython with efficient representations for small objects (such as tagged integers in pointers), the runtime need not allocate a `PyObject` for each argument and return value; it can use the efficient representation directly [16, 18].

8 Future work

Given how promising our early results are, we would like to build out support for more complex signatures, such as support for C strings and primitive types wider than 64 bits. We would also like to support a more expressive declaration language, such as the one used in the Argument Clinic.

In the future, we would like to see these type annotations automatically emitted by binding generators. Projects such as Cython and PyO3 already have machinery for generating wrapper code for C functions, and therefore have sufficient knowledge about the C function types.

Such binding generators also have more insight into the effects that happen inside a native function. For example, Cython may be able to statically guarantee that a function does not raise an exception, need to acquire the GIL, or something else. Reducing the set of effects from "all effects possible" could aid the PyPy optimizer.

Acknowledgments

To Sarah, for proposing the title. To Kate McKinnon, for ongoing comedic genius.

References

[1] [n. d.]. *HPy - A better C API for Python.* https://hpyproject.org/

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*. ACM, Montreal, Quebec, Canada. https://doi.org/10.1145/1297081.1297091

[3] Gergö Barany. 2014. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*. 1–9. https://dl.acm.org/doi/10.1145/2617548.2617552

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2010), 31–39.

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation removal by partial evaluation in a tracing JIT. In *PEPM*. Austin, Texas, USA.

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*. ACM, Genova, Italy, 18–25. https://doi.org/10.1145/1565824.1565827

[7] Carl Friedrich Bolz and Laurence Tratt. 2015. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming* 98 (Feb. 2015), 408–421. https://doi.org/10.1016/j.scico.2013.02.001

[8] Stefan Brunthaler. 2010. Efficient inline caching without dynamic translation. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 2155–2156. https://doi.org/10.1145/1774088.1774542

[9] Stefan Brunthaler. 2010. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, William D. Clinger (Ed.). ACM, 1–14. https://doi.org/10.1145/1869631.1869633

[10] Stefan Brunthaler. 2010. Inline Caching Meets Quickening. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 429–451. https://doi.org/10.1007/978-3-642-14107-2_21

[11] Stefan Brunthaler. 2021. Multi-Level Quickening: Ten Years Later. *CoRR* abs/2109.02958 (2021). arXiv:2109.02958 https://arxiv.org/abs/2109.02958

[12] Brandt Bucher. 2023. https://github.com/python/cpython/pull/113465.

[13] C. Chambers, D. Ungar, and E. Lee. 1989. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, Vol. 24. https://doi.org/10.1145/74878.74884

[14] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 20–33. https://doi.org/10.1145/3617651.3622982

[15] Antonio Cuni. 2018. Inside cpyext: Why emulating CPython C API is so Hard. https://www.pypy.org/posts/2018/09/inside-cpyext-why-emulating-cpython-c-8083064623681286567.html

[16] MicroPython Developers. 2024. smallint.h. https://github.com/micropython/micropython/blob/master/py/smallint.h.

[17] PyO3 Developers. 2024. The PyO3 user guide. https://pyo3.rs/v0.20.2/.

[18] Skybison Developers. 2024. objects.h. https://github.com/tekknolagi/skybison/blob/trunk/runtime/objects.h.

[19] Jake Edge. 2021. Making CPython faster. https://lwn.net/Articles/857754/

[20] Larry Hastings. 2024. Argument Clinic How-To. https://docs.python.org/3.10/howto/clinic.html.

[21] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. 2021. Static Type Inference for Foreign Functions of Python. In *32nd International Symposium on Software Reliability Engineering (ISSRE 2021)*. IEEE, 423–433.

[22] Greg Kroah-Hartman. 2005. container_of(). http://www.kroah.com/log/linux/container_of.html

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[24] Jukka Lehtosalo. [n. d.]. mypy - About. https://mypy-lang.org/about.html.

[25] Siliang Li and Gang Tan. 2011. JET: exception checking in the Java native interface. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 345–358. https://doi.org/10.1145/2048066.2048095

[26] Siliang Li and Gang Tan. 2014. Exception analysis in the Java Native Interface. *Science of Computer Programming* 89 (2014), 273–297. https://doi.org/10.1016/j.scico.2014.01.018

[27] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2022. Gradual Soundness: Lessons from Static Python. *The Art, Science, and Engineering of Programming* 7, 1 (June 2022), 2:1–2:40. https://doi.org/10.22152/programming-journal.org/2023/7/2

[28] Kate McKinnon and Colin Jost. [n. d.]. *Weekend Update: Dr. Wenowdis on Trump's Televised Health Exam - SNL*. Saturday Night Live. https://www.youtube.com/watch?v=2kQxVwYwrME

[29] Raphaël Monat. 2021. *Static type and value analysis by abstract interpretation of Python programs with native C libraries*. PhD thesis. Sorbonne Université. https://theses.hal.science/tel-03533030/document

[30] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 323–345. https://doi.org/10.1007/978-3-030-88806-0_16

[31] Russell Power and Alex Rubinsteyn. 2013. How fast can we make interpreted Python? *CoRR* abs/1306.6047 (2013). arXiv:1306.6047 http://arxiv.org/abs/1306.6047

[32] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[33] Yu-Hsin Tsai, I-Wei Wu, I-Chun Liu, and Jean Jyh-Jiun Shann. 2013. Improving performance of JNA by using LLVM JIT compiler. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. 483–488. https://doi.org/10.1109/ICIS.2013.6607886

[34] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[35] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–30. https://doi.org/10.1145/3485513