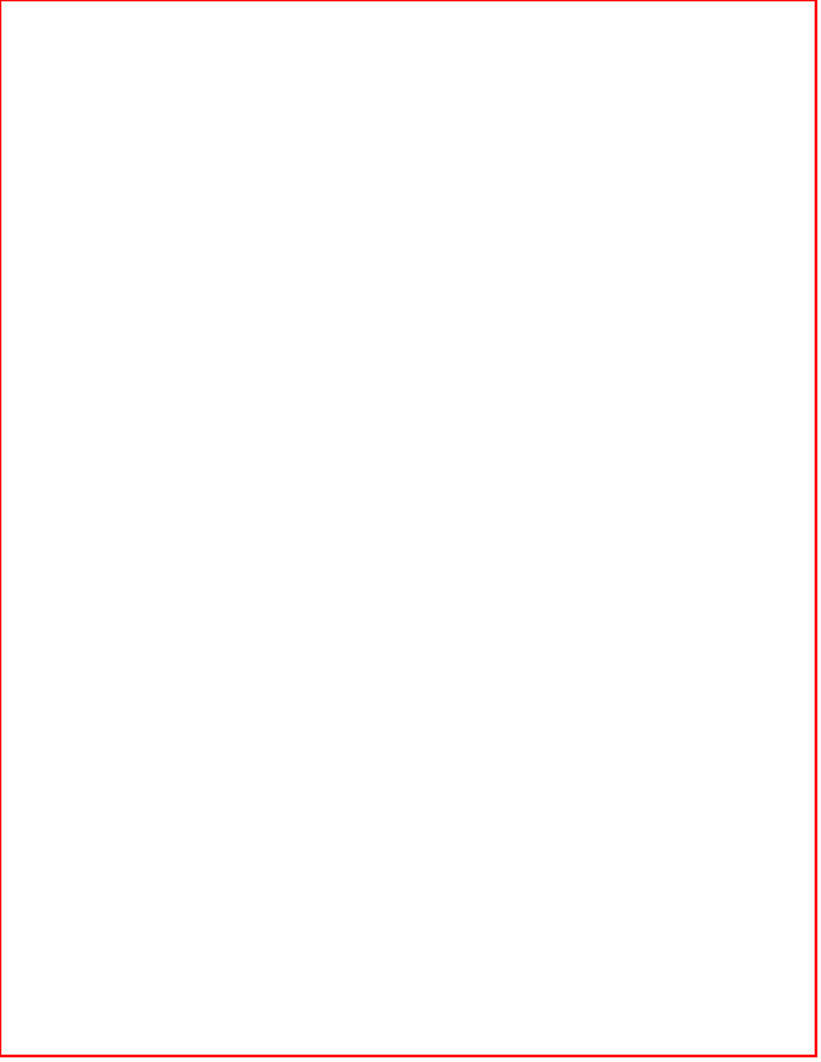

containers, are therefore automatically *pythonized*. Further pythonizations or reorganization of the presented library, is done in Python itself, rather than in an intermediate language. The PyPy JIT will inline such calls, leaving only a limited run-time cost.

Object/memory ownership is a long-standing problem of cross-language programming. Especially so in the case of Python-C++, because the Python programmer relies on reference counting (in CPython) or garbage collection (in PyPy). Fortunately, modern C++ provides standard templates (such as `unique_ptr` and `shared_ptr`) to express ownership rules. The Cling interpreter provides automatic template instantiation, leaving both the use of modern interfaces and memory management transparent to the Python developer.

The dynamic nature of Cling is well matched to the same of the Python interpreter. It is used to support cross-inheritance of Python classes from C++ classes, but also the other way around. It enables important optimizations such as iteration over STL's vector, and automatic downcasting of C++ objects on the Python side.

`cppyy` brings modern C++ bindings to PyPy, and the combination of Cling's dynamic nature with Python's and the dynamic optimizations of PyPy's tracing JIT bring new levels of performance and functionality.

This paper is organized as follows. Section II lays out the motivation for this work and provides historical context. Section III describes the architectural decisions that went into our module, and Section IV highlights a few of the more important features. We present micro-benchmark results in Section V and a discussion in Section VI. Finally, we briefly contrast our work with other tools and provide an outlook for future directions in Sections VII and VIII, respectively.

II. Motivation and Background

There are two major developments that have changed what is possible for Python-C++ cross-language bindings: changes to the C++ standard, and the availability of important components for building bindings generators.

The C++ language was standardized in 1998 and, other than a technical corrigendum in 2003, did not see any significant changes until 2011 when a new standard arrived. The C++ standards committee decided to keep up the pace of modernizing C++, introducing major revisions in 2014 and 2017, and planning yet another for 2020. Furthermore, there are enough technical proposals to keep this pace going well beyond that. The improvements in the language address major shortcomings in expressing semantic intent in interfaces. For example, when an Application Programming Interface (API) function returns a pointer to an object, the programmer has to resort to the documentation to find out whether the caller should take ownership or leave that to the callee. Automatic bindings generators suffer from the same problem, but can not read documentation. Sometimes, they can rely on convention, or take a clue from the function name. But more likely a programmer needs to intervene and mark the rules for individual functions. Since C++11, however, the intent can be clearly marked by returning either a `unique_ptr` or `shared_ptr`. The former takes ownership, the latter shares. Similarly, marking a method as `const` has grown the added meaning that it is safe to call that method concurrently with other `const` methods in a threaded environment. Exposing these semantics at the interface level, and thus accessible to the parsers of bindings generators, is hugely beneficial.

That still leaves the need for a parser that is up-to-spec for the most current C++ standards. Here we are in luck, due to the emergence of the LLVM project, and in particular its C++ front-end, Clang. Clang is Open Source, but most importantly extremely well documented and therefore easy to use. Even better has been the emergence of Cling, which is built on top of Clang, adding interactivity and dynamic execution to C++. This is a great match for a dynamic language such as Python. Lastly, we now have `cffi` for C, which adds fast interfaces to and from Python and C, for both the CPython and `pypy-c` interpreters. Although C++ has vastly more features and expressiveness than C at the language level, it is little different at the lowest levels for reasons of binary compatibility. Thus, we can use `cffi` also as the basis for C++ bindings to Python.

The work presented in this paper has integrated these existing, mature, technologies in `cppyy`. For the PyPy version, this took a little over 4K lines of RPython[2] and 2K lines of C++. For the CPython version, this took about 1K lines of Python and roughly 18K lines of C++.[3] As we will show throughout this paper, `cppyy` provides new, unique features. But the amplifier effect of integrating and reusing powerful tools has been an important ingredient for success.

There are three main goals that guided this work, and we consider each of them in detail in this section and the next:

- High performance of the bindings, good scaling, and easy distribution for large projects.
- Direct support for any and all C++ headers and transparent use of modern C++.
- Enabling new functionality by combining interactive Python with interactive C++.

A. High Performance

The number of scientific libraries that come with Python bindings is vast and it has long become possible for practitioners to do all their work in Python. The point where the performance of Python fails to be good enough is far out, but it is still hit too often. Common solutions to such performance problems are to rewrite parts of the Python code in another language, or to use language extensions. This is unfortunate, because it means extra work and often an extra learning curve. Furthermore, we have observed that teams of developers sometimes reject the use of Python at the outset, because of the expectation of having to redo work in a different language for performance reasons. Pushing back the point up

[2]RPython, or Restricted Python, is the implementation language of PyPy. It is best to think of it as the equivalent of what C is for the case of CPython, rather than to think of it as Python.

[3]A good chunk of C++ code is shared by both implementations.

to where Python performance is "fast enough" thus has a direct impact on the productivity of developers and the acceptance of the Python language.

For "Big data"-type analyses, it can be even more acute: these analyses tend to be largely I/O bound, caring little about getting the most out of the CPUs. However, with investments in non-volatile RAM and fast solid state disks, analyses risk becoming CPU-bound again because of languages like Python. Giving the Python portions of the analyses a performance boost can bring them back under threshold.

Finally, energy is fast becoming the main cost driver of supercomputers and large clusters. Improving the performance of Python, with full compatibility and without loss of functionality, reduces energy waste and thus cost.

B. Modern C++

From a usability perspective, the simplest possible automatic bindings generator is one that is pointed to the public header files of a C++ project, and figures it out from there. With modern C++ and semantic intent properly described in the header files this can be a reality. But this does require a modern C++ parser to go with it, which we get through the use of Cling. Of course, as new features get added to the C++ standard, they sometimes require explicit support in the bindings generator first. But more often, changes to the language have involved simplifications for the C++ developer (e.g. the keyword `auto` and vastly improved template support for generic programming). These changes impose greater requirements on the parser, but after canonicalization look no different at the AST level then code that the bindings generator can already handle, and are thus supported directly.

C. Interactivity

Having a C++ interpreter match up with a Python one, offers unique opportunities to hide the "C++ aspect" of bound code even more. These include automatic template instantiations, so that the Python developer need not consider whether APIs take `unique_ptr` or direct pointers, and need not "pre-instantiate" templates when generating the bindings to have the generated code linked in for use. It also means that templated code can be tried out and experimented with interactively.

It makes possible to (interactively) derive Python classes from C++ ones for use in C++ component frameworks, which work through pre-defined abstract base classes. In fact, we can even do the opposite: derive C++ classes from Python ones, and use the result in Python.

In this paper, we focus mostly on performance, but we consider the improved interactive experience just as important.

D. Historical Context

The current, Cling-based, `cppyy` project for PyPy originates from an earlier effort[6] based on `gccxml`[7] and Reflex[8]. Cling surpasses Reflex by providing a fully compliant, interactive C++ interpreter. It brings support for modern C++, it greatly improves ease of use, and it allows dynamic optimizations. The Reflex-based work itself originates from earlier work for CPython, PyROOT[9], done in the context of the ROOT[10][11] project. PyROOT did not only provide Python bindings for ROOT, but also for many core frameworks, reconstruction, and analysis codes for High Energy and Nuclear Physics experiments. Many of the design ideas of `cppyy` have grown out of almost a decade and a half of practical experience of Python-C++ bindings for large projects.

III. Design

The design of `cppyy` originated from a desire to painlessly introduce Python in an environment dominated by huge, widely used C++ codes. There was a chicken-and-egg problem because C++ developers would not spend the effort to make Python bindings available until users materialized, but users could not get started until they had bindings to use. Therefore, the emphasis has always been on fully automated bindings generators. The architecture of `cppyy` is fully generic to fulfill this automated role, with specializations for performance, specific use cases, and to match low-level features.

Internally, `cppyy` holds "converters" to turn Python objects into C++ ones and vice versa. Likewise, it has "executors" for running functions and producing Python results. Converters and executors are specialized for all known builtin types and for some common C++ classes (such as `string`). There are generic ones to cover user-defined C++ types.

With these as a basis, more complex proxies for C++ constructs can be build: a single function holds a list of converters, one for each argument, and a single executor. An overload holds a list of functions. A class holds a list of overloads, a list of converters (one for each data member), and has a set of other classes as bases. A namespace contains lists of classes, functions, and data members. Selection of the proper converters and executors is done based on type names, which are represented as strings and thus language-agnostic, to keep this build-up mechanism fully generic.

C++ is a language full exceptions, however, and compilers/linkers are allowed to break language rules as long as there are no visible side effects to C++ programs. There are then a few cases that fall outside the mold described above. For example, although inline functions are supposed to have external linkage, linkers tend to remove them if there are no out-of-line uses. Other examples include redefined `operator new` and `operator delete`, and default arguments of structs or classes that are passed by value. To cover such cases, wrapper code with predefined interfaces is written and JITed (through LLVM) by Cling. Wrapper code includes generic function wrappers, casting functions for diamond-shaped class hierarchies, and automatic templates. Because such wrappers are compiled by a true C++ compiler, these exceptional cases are automatically taken care of.

A. Scale

Cling was designed with large scale and distribution in mind: many thousands of C++ classes across multiple projects. `cppyy` retains that by utilizing Python's dynamic nature.

First, virtually all bindings are generated dynamically: Python-side proxy classes are not constructed until access and function argument converters not created until called. This interferes somewhat with the explorative nature of interactive Python, but that is mostly alleviated with custom `__dir__` methods and doc strings, that return lists of strings and information about arguments, without actually constructing the corresponding objects.

When Python code accesses a class, or other C++ entity, through its containing module (either the global or a named namespace), Cling is requested to load the needed definitions. These can live in (text) header files, precompiled headers, and soon also in C++ modules.[4] The latter is the most memory-efficient approach, as it allows deserializing only the minimum necessary subset of the module (headers, as text or precompiled, are loaded en-bloc).

Second, the only part that uses the respective Python interpreter APIs is in the `cppyy` module. Given the large differences between CPython and `pypy-c`, there are two versions of `cppyy`, but it only takes a recompilation (retranslation) of that to create a module for different versions of an interpreter. This means that a C++ project can ship one set of precompiled headers/modules[5] and let it up to the Python developer which interpreter to use.

Third, all C++ classes, functions, variables, etc. are loaded underneath `cppyy.gbl`, the global namespace. This means that if two projects want to use the same C++ class, the Python developer actually sees the same Python proxy class in both, allowing inspection such as `isinstance` and `issubclass` to work as designed in all cases. Contrary to CPython's approach to extension modules, this does mean that name clashes are possible. However, the chance of that is rather remote, because if a clash would occur in this way, it would already happen when linking in the C++ libraries, meaning that these could not have been used together in the first place.

B. Presentation

The presentation of all C++ classes in their natural C++ location has another advantage when making a large C++ code base available to Python for the first time: documentation for C++ use can readily be applied to Python use as well. In addition, questions to the original C++ developer are easily formulated in their C++ use, in case the original developer is not familiar with Python.

Although automatic pythonizations are available, in all cases C++ constructs work as well. For example, loops over STL collections can use the normal Python `for` loop syntax, or be coded using `begin()`/`end()` calls and use of C++ iterators. Developers can register callbacks to further pythonize their classes. Callbacks are used to retain all benefits of lazy loading. Alternatively, a new interface can be designed in Python, directly on top of the bindings.

[4]Experimental support exists, but modules are due for C++17 at the earliest.

[5]Caveat: if these contain system headers, they need to match, or the local system headers need to be preloaded, to override settings in the modules. For widely used operating systems, we know which they are and do the necessary preloading automatically.

IV. Features

Both the Python and C++ languages are still evolving, and so a language binding between them may never be feature complete. Furthermore, some corners of the C++ language cannot be resolved without programmer intervention (e.g. overloads between `bool` and `int`, or the use of `char*` in the role of "`byte*`"). Nevertheless, we believe the implementation to be very complete and there are ~1200 feature unit tests to back up that assertion. In general, most features are implemented in the most natural way. For example, C++ class hierarchies are fully duplicated on the Python side to allow introspection to work, namespaces act like modules, data members are directly addressable, etc. A listing of features going into quite some detail can be found in [12]. Here we will highlight some of the more interesting ones.

A. Automatic Templates

The ability to automatically instantiate templates is particularly useful for handling modern interfaces, for example to transparently use `unique_ptr` and `shared_ptr`, as already explained above. But it is essential to make class method templates useful, and greatly helps the use of STL containers, the utility classes of which (such as pairs and iterators) are all templated.

As an example, consider this interactive session:

```
>>>> import cppyy
>>>> cppyy.Declare("""class A {
.... public:
....     A(int i=42) : m_int(42) {}
....     int m_int;
.... };""")
True
>>>> from cppyy.gbl.std import vector
>>>> from cppyy.gbl import A
>>>> v = vector(A)(10)
>>>> print v[5].m_int
42
>>>>
```

Note the interactive creation of the C++ class A, and the instantiation of the STL vector with it, which is subsequently ready for use.

B. Automatic Downcast and Object Identity

The Python language is strongly typed, but does not have the concept of casts. To avoid casts, C++ objects on the Python side are automatically cast down to the most derived known[6] type in its class hierarchy. Casts within a single inheritance hierarchy are simple, as these are fixed offsets (and are most commonly zero). In the case of inheritance diamonds, caused by multiple virtual inheritance, Cling generates and JITs a casting function, which takes a pointer to the instance. This is

[6]Component libraries typically only expose abstract base classes.

called to get the offset specific to the instance. The casting function is memoized, and so is the offset specific to the instance when JITed by PyPy in a trace.

Auto-downcasting serves a second purpose. When objects are passed through one base class as argument and returned (not necessarily by the same call) through another, their identity might be lost. Having multiple Python-side objects point to the same C++ object at different offsets makes memory management much more difficult. With auto-downcasting, object hashes can be stored by type with a weak reference, allowing "recycling" of Python objects and thus preservation of object identity.

C. Cross-language Inheritance

The dynamic nature of Cling allows injection of C++ classes into "dynamic scopes." Thus we can create C++ representations of Python classes. After inspecting the `func_code` data member of functions, methods are filled out with generic templated arguments, that are specialized where necessary. The arguments internalize the type on instantiation (i.e. when the function is called), to create `PyObjects` out of them. These are forwarded to the actual Python function as arguments. Since the class is a true C++ class, it can serve as a base class on the C++ side, but of course only dynamically. Such a derived class can hide the methods in the base by overriding them and be transferred back to Python. Auto-downcasting and the method resolution order (mro) of Python make sure that the overriding works there as well.

The opposite direction is much more useful: after deriving a Python class from a C++ class, Python instances can participate in C++ component frameworks. For this to work, a vtable needs to be created and this is done using an intermediate C++ class, generated by the Python-side meta class. The meta class inspects the list of methods in the interface and creates dispatchers for each. Because C++ is statically typed, all types are known from the interface. The Python side class constructor is augmented by the meta class to call the intermediate class, passing `self`. Upon call of a dispatcher method, it finds the necessary method in the dictionary of `self`. By doing the lookup each time (as is done in Python), normal Python dynamic behaviors are retained.[7]

Note that in both cases of cross-inheritance, we make use of the CPython API through the `cpyext` module, which bridges the C-API with PyPy internals, but is not optimal for this use case. Multiple inheritance is not yet supported.

D. Pythonization and Pythonization Rules

Generating automatic bindings is fast and easy, but the result isn't always pythonic. Developers can pass callbacks that implement pythonization rules. The callback, written in Python, searches for patterns and wraps calls that match. For example, functions that are found to take a naked pointer and a size (together representing an array) can be wrapped to take a Python array. Upon call, the wrapper splits the array into two arguments, the buffer and its size, and passes them individually to the actual function.

We implement a few generic rules inside `cppyy`. For example, any class that has STL-like `begin()`/`end()` methods returning iterators, is automatically modified to implement the Python iterator protocol. There are many specialized pythonizations for STL, such as slicing of vectors, iteration over pairs, and conversions between string objects.

We provide a set of flags on methods that fine tune their behavior with specific ownership rules, threading policy, signal policy, and exception handling. A Pythonization rule can be build on the function name if the C++ project follows strict coding conventions. For example, all functions that are named `CreateXYZ` can be assumed to return a new object for which the caller needs to take ownership.

As explained in the introduction, modern C++ expresses intent more clearly, thus we expect the importance of pythonization rules to diminish.

V. Benchmark Results

We test our implementation using micro-benchmarks. Since we are primarily interested in the performance of the bindings, they are written to spend maximum time in the bindings, not in processing on the Python or C++ side. Clearly, in actual, practical use, developers will minimize the time spent in the bindings, meaning that relative overheads will be smaller than reported here when measured for end-to-end applications. We use `pytest-benchmark`[13] for running the Python micro-benchmarks. Warmup is enabled, but otherwise we stuck to the default options.

We do not release the Global Interpreter Lock (GIL) inside the loop body, because the PyPy JIT determines at run-time by the presence of threads whether to release or not, thus not releasing in the other benchmarks either allows for an apples-to-apples comparison. The default for both `cppyy` and SWIG is to *not* release. The "warm-up time" of the PyPy JIT and Cling wrapper generation is subtracted by `pytest-benchmark`. This is a real cost, but it is a one-off. Therefore, its relative contribution to the total time can be made arbitrarily low by simply increasing the duration of the benchmark, or by reducing the trip count threshold before JITing starts to zero. In practice, the "warm-up time" depends on the complexity of the code. However, it is safe to assume that larger, complex codes also take longer to execute, thus keeping the relative cost low.

When comparing with CPython, we use primarily the CPython `cppyy` module as that is equal in functionality. Where possible, since it supports only a smaller subset of the C++ language, we compare with `SWIG` v3.0.10[14], a well known and widely used fully automatic bindings generator. The C++ reference codes are all compiled with g++7.3, optimized using -O2, and reported numbers are averaged over 5 runs. Timings for PyPy are provided both for the FFI and the wrapper path.

We have chosen the benchmarks to reflect common uses that form the (performance) basis of more complex functionality.

[7] With the exception of replacing `__init__` of course, as that would remove passing `self`.

For example, because of auto-downcasting and the eliding of offset calculations in the JIT, there is no cost difference between virtual and non-virtual functions. Similarly, once instantiated, there is no difference between templated and non-templated methods. Work remains to be done, however (e.g. object by-value return is not supported on the fast path), but the `cppyy` module is designed to fall back on the wrapper path when the fast path is not available. The wrapper path is suboptimal for our purposes, but as we will show, it is still a lot faster on `pypy-c` than its equivalent on CPython.

A. Instance Methods

The first base we consider are non-overloaded instance methods: we create an instance and call a method in a loop. Here, the method takes an `int` and returns an `int`. The function body adds a data member (also an `int`) and returns the sum of it and the argument passed. Calls from C++ into C++ are from a main program into a shared library. This means that for non-virtual functions, the call will go through the Procedure Lookup Table (PLT); and through the vtable pointer for virtual functions. This is slower than direct calls, and does not allow inlining, but is more representative of actual use of C++ in large projects.

The results are shown in Figure 1. The handling of the GIL dominates the `pypy-c` FFI path. When not releasing the GIL, the overhead is still about $4x$ over C++: there are checks (called *g*uards) in the trace, e.g. to see whether the "`this`" pointer has changed or whether the integer return has overflown (to set a Python OverflowError exception if needed). The wrapper path, which is still optimized by the PyPy JIT, is considerably slower than the FFI path, but easily beats either of the CPython approaches. CPython/cppyy outperforms SWIG by a good margin in this simple case, because SWIG has a Python intermediate layer that forwards the calls, which adds a lot of overhead if little time is spent in C++.

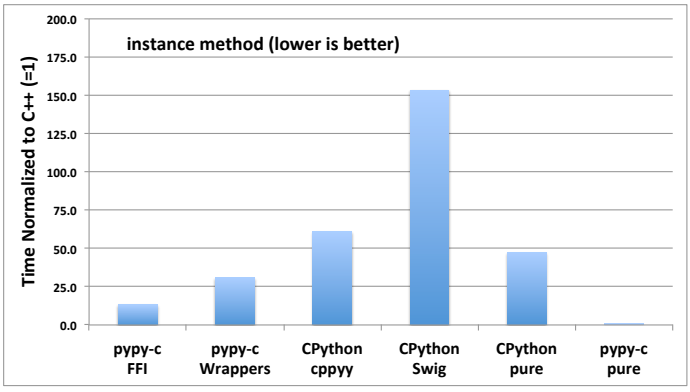

Fig. 1: *Relative performance (compared to C++) of the FFI and wrapped paths for instance method calls. CPython using cppyy, SWIG, and pure Python calls are shown for reference.*

For reference, we also show the results of pure Python code on CPython. The same result for `pypy-c` is equal to C++ (we prevented both from inlining the call, to make sure a call actually occurred: either could optimize the empty body/loop out of existence).

To find out what this level of overhead means in practice, we replace the method implementation with a less trivial body: a call to (`math.`)`atan` on the argument is added. The results are in Figure 2. The relative standing remains similar, but as can be seen the total relative overhead drops quickly. In fact, If we remove handling of the GIL, the overhead for the FFI path disappears in the noise. The pure Python code now has an external call as well, and slows down considerably. Running `pypy-c` with pure Python benefits from knowing the function specification and runs at $1.8x$ of C++, or almost twice as fast as the generic FFI path.

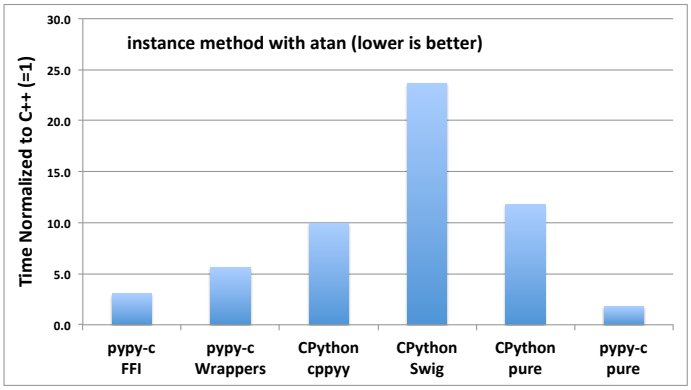

Fig. 2: *Relative performance (compared to C++) of the FFI and wrapped paths when calling a non-trivial function. CPython using cppyy, SWIG, and pure Python calls are shown for reference.*

The next base case uses overloaded instance methods, and the results are shown in Figure 3. There is no difference between an overloaded or non-overloaded method in C++ as the overload resolution is at compile time. In the case of `pypy-c`, there is only a very minor slowdown. All Python codes resolve overloads at run-time. First, functions are sorted based on the possibility of implicit conversions for their argument types, with the more restricted types given higher priority. For example, when overloading a method that takes an `int` with a method that takes a `double`, the former needs to be tried first, as all integers can be converted to floating point, but not vice versa. When such an overload is then called with a `float` argument, it will fail the first method and then (correctly) succeed the second. Conversely, calling with an `int` will succeed in the first method, never (again correctly) reaching the second.

In the case of `pypy-c`, after the JIT collects an execution trace, the optimizer will remove all paths that are guaranteed to fail, replacing them by pre-condition checks. Thus, in the above example, if a floating point argument is used, only the second call will remain in the trace. Similarly, if an integer is used, only the first call will remain. Whereas trying (and failing) a conversion can be expensive, checking a pre-condition is cheap and its additional overhead tiny. The net

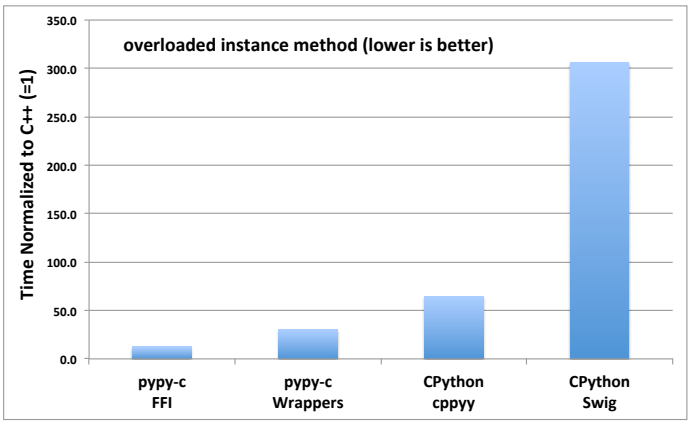

Fig. 3: *Relative performance (compared to C++) of the FFI and wrapped paths for overloaded instance method calls. CPython using cppyy and SWIG are shown for reference.*

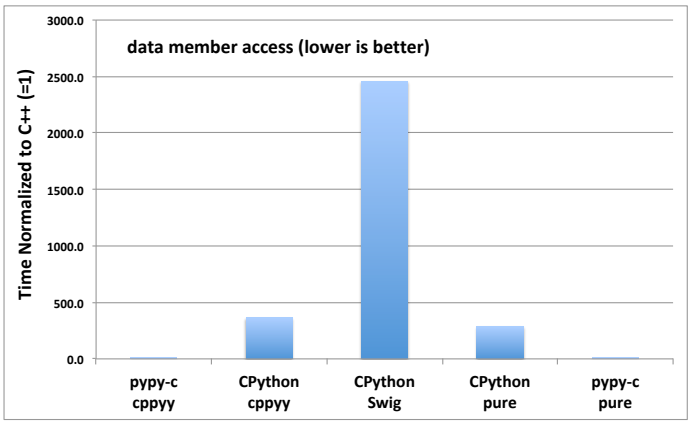

Fig. 4: *Relative performance (compared to C++) of data member access in* `pypy-c`. *CPython using cppyy, SWIG, and pure Python are shown for reference.*

result is that, after "warm-up," the performance of overloaded and non-overloaded methods is the same.

SWIG and CPython/cppyy perform overload resolution in a similar manner, but the latter also memoizes the selected overload based on a hash of the argument types and selects the correct overload if the same types are seen on the next iteration. Although both have an increased overhead for overloaded than non-overloaded functions, memoization clearly outperforms.

B. Data Member Access

Access to data members is relatively straightforward. It involves only an offset calculation from the `this` pointer of the C++ object. `cppyy` supports all cases, including static data, global pointers, and multiple virtual inheritance. What is left, is the mapping from Python types to C++ types and vice versa. This, too, is fully supported, including calls to `operator=` when assigning by value. We implement a micro-benchmark that swaps the values of the data members of two instances, thus testing data member lookups, reads and writes. The results are plotted in Figure 4.

We do not show separate numbers for the wrapper and FFI paths, because data access does not involve function calls (unless an assignment operator is used). `pypy-c` makes all accesses directly to memory. It is only $1.7x$ slower than C++ with `cppyy` and only $4.2x$ slower for pure Python. The latter is slower because it retains more guards: in C++ instances, data members are fixed and at fixed offsets. Thus, only the `this` pointer needs guarding, whereas the pure Python case also needs guarding against changes to the instance.

All CPython cases are considerably slower. SWIG comes in at such low performance because it implements C++ data members with properties in Python. While `cppyy` does the same, its properties are coded up in C++. The pure Python version uses data members, not properties.

C. Standard Template Library vector

The STL vector is such a common container that it is worthwhile to explicitly optimize for it. The C++11 standard guarantees that a vector stores its payload as contiguous memory, and provides access to it through the `data()` method. The standard also disallows modifying a vector in a loop (in fact, all outstanding iterators are invalidated after modification[8]). Thus, we can create a vector-specific iterator that calls `data()` at the start, keeps an index, the element type size, and a type-specific converter. When iterating, access is the same as for data members: the base and offset are passed to the converter, which returns a Python object. The results are presented in Figure 5.

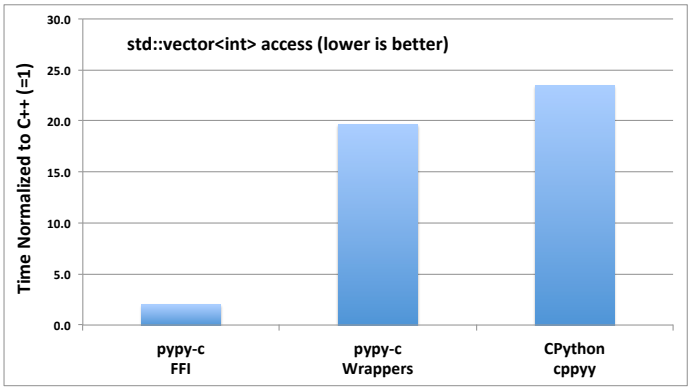

Fig. 5: *Relative performance (compared to C++) of looping over an STL vector in* `pypy-c` *and CPython using cppyy. SWIG not shown: it is* $\sim 350x$ *slower.*

Instead of looping explicitly over the vector, we used Python's builtin `sum` method, so the iteration is on the C-side and spurious dictionary lookups are avoided. Thus, after initial setup, which establishes start, end, and step size of the

[8]It is straightforward to set a flag in all vector modifying methods. Iterators could then check for modification and raise an exception if needed. We have not yet implemented this.

vector, the only calls into `cppyy` remaining are the converter calls. Although both CPython/cppyy and `pypy-c` with the wrapper path are rather slow compared to C++, which fully inlines the vector operations, `pypy-c` comes close at only $2.5x$ slower. SWIG performs `getitem` calls during iteration, with the expected very costly overhead.

D. Basic Analysis Code

Our final micro-benchmark runs a simple "analysis-style" code: it produces some data (using a random number generator), does a small bit of arithmetic on them in Python, and stores the resulting values in C++ objects. During the loop and at the end of the run, these C++ objects are serialized and written to disk. The code contains both a loop with a large trip count that `pypy-c` can optimize, but also plenty of prologue (initialization, C++ object creation) and epilogue code (storing to disk). It is representative of analyses in High Energy Physics. The results are shown in Figure 6. SWIG is missing because classes are used that it can not parse.

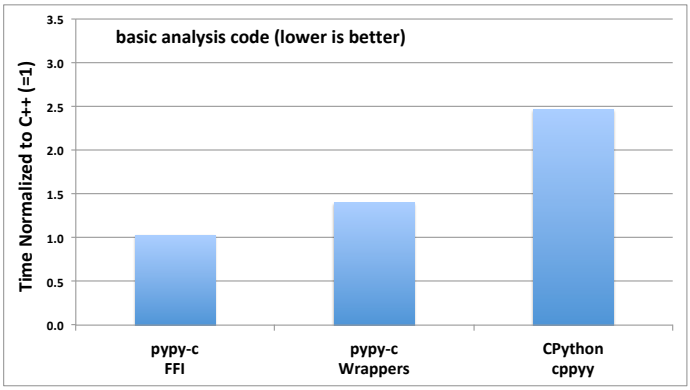

Fig. 6: *Relative performance (compared to C++) of a basic analysis code in* `pypy-c`. *CPython using cppyy is shown for reference.*

The benchmark runs a lot more than just bindings, with some work done on the Python side, but with enough time spent in C++ that it should dominate. Still, the cost of the Python-C++ bindings is large enough in the CPython case for it to be $3.3x$ slower than C++. Most of the time is spent in the conversion of floating point values, used as function arguments. Both the wrapper and FFI paths improve the performance considerably, with the latter running only 10% slower than C++, or $3x$ faster than CPython. Even when CPU time is considered "free," a difference of a factor of 3 can affect productivity greatly as turnaround time matters when prototyping.

VI. Discussion

The `cppyy` module is able to vastly reduce time spent in Python-C++ bindings when used from `pypy-c`, compared to equivalent codes on CPython. Speed-ups come from the PyPy JIT, which optimizes and compiles loop bodies that exceed the (user settable) trip count. Non-JITed code actually runs about $2x$ slower than CPython and this is also roughly true for code using `cppyy`, because the top-most interface is pure Python. Calling the underlying interface directly is "only" as slow as CPython/cppyy (there is little difference for JITed code, with the direct calls being ~15% faster). However, scientific codes tend to spend most of their time in loops. Thus, although work should be done to bring `pypy-c` more in line with CPython for non-JITed code, this is not a significant limitation in practice for the targeted audience.

The PyPy JIT is designed to optimize Python code, but `cppyy` is written in RPython in order to allow low-level features, such as pointer arithmetic. Such code is not always a good fit and we had to make sure that the code paths seen by the optimizer are as straightforward as possible. For example, classes in a single inheritance hierarchy are presented differently than those in a multiple virtual inheritance one. Doing so reduces the number of necessary run-time checks, and thus the number of branches in the traces. The optimizer can then be more aggressive, resulting in better performance. Note that PyPy provides direct support to test JIT behavior. This enables unit testing of detailed specializations that are written for the express benefit of the JIT, to catch performance regressions early.

The emphasis of the micro-benchmarks was on performance. However, even at such small scale it was clear that parsing header files directly is far more user-friendly than working with an intermediate language.

VII. Related Work

There are several automated Python-C++ bindings generator tools available, for example SWIG[14][15] and SIP[16]. SWIG can generate bindings for several dynamic languages, Python being only one of them. Other libraries, such as e.g. boost.python[17], ease the integration of C++ features such as namespaces, overloading, and exception handling by providing a C++ API for that purpose.

All these tools, including our `cppyy` module, have in common that there are no extensions to the Python language used or restrictions imposed of what Python language constructs are allowed/supported. All except `cppyy` use the Python C-API and require linking with the Python libraries. SIP and `cppyy` disambiguate identical classes used by different projects and keep the structure of C++ namespaces. Both also support lazy loading/instantiating of Python classes and methods. SIP and SWIG are automated tools, but require an intermediate specification as these projects have developed their own parsers, which do not support the full C++ language. To alleviate this problem, an external parser, such as Clang, can be used to generate specification files that are restricted to the feature sets understood by SWIG or SIP, respectively. The SWIG specifications are not limited to Python and can be reused for any of its multiple backends.

Our approach does not use any intermediate specification or extension, so that no additional language beyond Python (and some C++) needs to be learned to use `cppyy`. There are two versions of `cppyy`: one written in RPython for PyPy and one written in C++ for CPython. Thus only the `cppyy`

module is specific to any given (version of) Python interpreter, which greatly simplifies distribution of large codes, as well as its reuse in other large projects. Loading of namespaces and classes is automatic and lazy in `cppyy`, again for reasons of scale and distribution. We repurpose the Cling parser, thus future-proofing our work. Having the full AST available also opens opportunities for dynamic optimizations.

Unique to `cppyy` is the use of the Cling interpreter to reduce the impedance mismatch between static C++ and dynamic Python, allowing for interactive instantiations of templates, cross-language inheritance, etc.

VIII. Conclusions

In this paper, we described `cppyy`, a new module for `pypy-c` that combines Cling, `cffi`, and PyPy's toolbox to deliver Python bindings to modern C++ with high performance. Modern C++ allows better expression of intent in interfaces, greatly facilitating automatic bindings generators. For example, by clearly indicating ownership and thread-safety. The `cppyy` module was designed with scale and distribution in mind: it constructs bindings lazily and has greatly reduced dependencies on the Python interpreter. The use of interactive C++ goes a long way to remove the impedance mismatch between dynamic Python and otherwise static C++. We were able to show an order of magnitude improvement in performance for function calls, and two orders of magnitude for data access. Overall, this should result in a $\sim 3x$ speedup end-to-end in practical use.

Optimizations were made possible by deconstructing high-level concepts to low-level interfaces. But the PyPy optimizer is designed to deal with higher level constructs, albeit Python ones. When C++ modules are fully supported, it will be possible to annotate functions, expressing higher level information such as the existence and scope of side effects, whether pointers to arguments are taken, or whether global data is accessed. Having such information available to the PyPy JIT would allow it to make less conservative decisions and optimize at a higher level: e.g. decide whether to release the GIL, elide a function call, or change a data structure.

We therefore expect, having proven the fundamentals, that our work can enable a whole new range of Python-C++ cross-language optimizations.

Acknowledgment

This work was made possible by the ATLAS Collaboration, Google Summer of Code and CERN SFT.

References

[1] PyPy project. [Online]. Available: http://pypy.org
[2] cffi documentation. [Online]. Available: https://cffi.readthedocs.io
[3] Cling. [Online]. Available: https://root.cern.ch/cling
[4] clang: a C language family frontend for LLVM. [Online]. Available: http://clang.llvm.org
[5] The LLVM Compiler Infrastructure. [Online]. Available: http://llvm.org
[6] cppyy documentation. [Online]. Available: http://doc.pypy.org/en/latest/cppyy.html
[7] GCC_XML. [Online]. Available: http://gccxml.github.io
[8] Reflex. [Online]. Available: https://root.cern.ch/how/how-use-reflex
[9] J. Generowicz, W. Lavrijsen, M. Marino, and P. Mato, "Reflection-based Python-C++ bindings," in *Proceedings of Computing in High Energy and Nuclear Physics*, September 2004. [Online]. Available: http://www.osti.gov/scitech/biblio/835826
[10] I. Antcheva, et. al, "ROOT – a c++ framework for petabyte data storage, statistical analysis and visualization," *Computer Physics Communications*, vol. 180, p. 24992512, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509002550
[11] ROOT data analysis framework. [Online]. Available: https://root.cern
[12] cppyy documentation. [Online]. Available: http://doc.pypy.org/en/latest/cppyy.html#features
[13] pytest-benchmarks documentation. [Online]. Available: http://pytest-benchmark.readthedocs.org/en/stable/
[14] SWIG project. [Online]. Available: http://swig.org
[15] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with c and c++," in *Proceedings of The 4th Annual Tcl/Tk Workshop*, July 1996. [Online]. Available: http://www.swig.org/papers/Tcl96/tcl96.html
[16] SIP project. [Online]. Available: https://riverbankcomputing.com/software/sip/intro
[17] boost C++ libraries. [Online]. Available: http://www.boost.org