High-performance Python-C++ bindings with PyPy and Cling

Updated August 24, 2018; see also: http://cppyy.readthedocs.io/en/latest/
Performance results collected for cppyy1.2.4, with PyPy2 6.1 (pre-release), and Python3.7

Wim T.L.P. Lavrijsen
Lawrence Berkeley National Laboratory
Berkeley, California
Email: WLavrijsen@lbl.gov

Aditi Dutta
Nanyang Technological University
Singapore
Email: aditidutta137@gmail.com

***Abstract*—The use of Python as a high level productivity language on top of high performance libraries written in C++ requires efficient, highly functional, and easy-to-use cross-language bindings. C++ was standardized in 1998 and up until 2011 it saw only one minor revision. Since then, the pace of revisions has increased considerably, with a lot of improvements made to expressing semantic intent in interface definitions. For automatic Python-C++ bindings generators it is both the worst of times, as parsers need to keep up, and the best of times, as important information such as object ownership and thread safety can now be expressed. We present `cppyy`, which uses Cling, the Clang/LLVM-based C++ interpreter, to automatically generate Python-C++ bindings for PyPy and CPython. Cling provides dynamic access to a modern C++ parser and PyPy brings a full toolbox of dynamic optimizations for high performance. CPython is supported for compatibility. The use of Cling for parsing, provides up-to-date C++ support now and in the foreseeable future. We show that with PyPy the overhead of calls to C++ functions from Python can be reduced by an order of magnitude compared to the equivalent in CPython, making it sufficiently low to be unmeasurable for all but the shortest C++ functions. Similarly, access to data in C++ is reduced by two orders of magnitude over access from CPython. Our approach requires no intermediate language and more pythonistic presentations of the C++ libraries can be written in Python itself, with little performance cost due to inlining by PyPy. This allows for future dynamic optimizations to be fully transparent.**

I. Introduction

The Python programming language and its standard interpreter allow easy integration with codes written in other languages. This has been an important condition for Python to establish itself as a high level productivity language for use with scientific codes. Python's run-time performance is very slow compared to other languages, such as Fortran, C, and C++. However, when used in conjunction with high performance libraries, this is of little consequence: the bulk of the time is spent in those libraries, rather than in the Python interpreter. Thus, the impact of Python on end-to-end performance is generally only small.

The availability of good Python bindings has led to an increasing number of scientists who program exclusively in Python. And with more code residing in Python, its performance becomes more important. The PyPy project[1] provides a highly compatible, alternative Python interpreter (`pypy-c`), that is designed for speed. PyPy achieves speed-ups by using a tracing just-in-time compiler (JIT) that dynamically specializes on the Python code executed, and compiles the result down to machine code. With PyPy comes a new project, `cffi`[2], which provides a fast foreign function interface (FFI) to the C language for both `pypy-c` and the original Python interpreter (from here on referred to as "CPython").

We are interested in bringing the same performance to PyPy for C++ as `cffi` provides for C. The important insight is that C++'s promise of "only paying for what you use" means that after high level language constructs are broken down, what is left is as simple as C. Once there, `cffi` can readily be applied. To break down C++, we need access to the abstract syntax tree (AST). For this, we use the Cling C++ interpreter[3], which is based on Clang/LLVM[4][5]. Using a C++ interpreter, rather than a static description of the AST, is important to reduce impedance mismatches with the dynamic nature of Python.

We provide a Python module, `cppyy`, that combines Cling and `cffi` for PyPy.[1] We provide two binding paths: one based on wrappers, which are JIT-ed by LLVM, and another based on `cffi`. The former is necessary to guarantee support for the full C++ language. The latter is applied strictly only then when preconditions are met, guaranteeing proper execution of the binding. Otherwise, it falls back to the wrapper path.

Cross-language access to data is comparatively simple and can readily be inlined by the JIT after adding checks to ensure that data has not moved. In `cppyy`, we support all cases efficiently, from simple global pointers to public instance data of classes in a multiple virtual inheritance hierarchy.

The interface presented to the Python developer is a global namespace (`cppyy.gbl`) from which C++ entities can be imported. This is well structured and scales readily, as all bindings generation can occur lazily and all Python-side classes are uniquely identified. It is, however, not very pythonic. Known constructs, such as Standard Template Library (STL)

[1]We also have an implementation of `cppyy` for CPython (which does not integrate `cffi`); see Section V for performance comparisons. For either interpreter, `cppyy` is available from: https://pypi.org/project/cppyy/.

PyHPC2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-5220-2/16/$31.00 © 2016 IEEE