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

Wim Lavrijsen (LBNL) and Aditi Dutta (Nanyang Tech)

PyHPC 2016
6th Workshop on Python for High-Performance and Scientific Computing
November 14, 2016, Salt Lake City, UT, USA
Background: High Energy Physics

• High energy physics (HEP)
  - A.k.a. “particle physics”, explores matter, energy and the fundamental forces of nature
  - Often works on huge, long running experiments in large, geographically dispersed collaborations
  - The original “Big Data”

• Software development challenges
  - Range of different skill sets, preferences, interests
  - Large turnover of people over experiment life time
  - Run on everything, everywhere: grids, clusters, HPC systems, clouds, and @home
Background: Python in HEP

• Historical time line of Python usage
  - 2001: first interest and implementations
  - 2004: gone mainstream
  - 2009: drives frameworks, job transforms, analyses
  - 2013: Nobel Prize in Physics (Higgs boson)
  - 2016: first-class citizen in new experiments

• Technology
  - C++ adopted in 1994, main language since ~1998
  - Python bindings home-grown: piggy-backed on C++ reflection for serialization and interactivity (CINT)
  - Increased Python use thanks to Machine Learning
$H \rightarrow ZZ \rightarrow 2e2\mu$
Our Goals

- Support C++11 and beyond
- The scale and distribution to support large codes
- High performance (with PyPy)
First Target: C++11 and beyond

- C++ language standardization went hyperdrive
  - Then: C++98
  - Now: C++11, C++14, C++17, C++2x, ...

- We parse C++ headers for Reflection to
  - Automate I/O and schema evolution
  - Use C++ interactively from an interpreter
  - Provide automatic Python-C++ bindings

- Impossible to keep up with a small team ...

CINT, a homegrown parser originated at HP, was replaced with Cling, which is an interactive C++ interpreter based on Clang (LLVM). Cling is developed by CERN. Our CPython-based Python bindings have followed suit.

\(^1\text{With technical corrigendum in '03}\)
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ developers, but Python users</td>
<td>Fully automatic, interactive bindings, based on parsing C++ headers</td>
</tr>
<tr>
<td>Huge number of classes, functions, etc.</td>
<td>Lazy lookup/creation: pre-compiled modules and bindings only at run-time</td>
</tr>
<tr>
<td>Lots of libraries and dependencies</td>
<td>Automatic loaders with search paths</td>
</tr>
<tr>
<td>Name clashes, duplicates</td>
<td>Follow C++ (i.e. linker) structure to scope and uniquely identify names</td>
</tr>
<tr>
<td>Too much “C++ feel”</td>
<td>Reflection-based pythonizations (automatic) and regexp-based support for pythonizing common patterns</td>
</tr>
<tr>
<td>Different Python versions: v2, v3, CPython, pypy-c, ...</td>
<td>Only core bindings module (cppyy) depends on Python</td>
</tr>
</tbody>
</table>

We leverage Python's and Cling's dynamic natures to maximize lazy evaluation, leading to shorter startup times and lower memory use.
Third Target: High Performance

- Important for perception and decision making
  - Python is not slow for those who use it
    - Part truth (heavy CPU loads in C++), part self-selection

- Improve performance completely transparently

Note: this turns out to be a rather small, and changing, group of Python users!
Technologies (Re-)Used

- **Goal:** maximize reuse of existing projects
  - Capture expertise, maintenance, future development

- **Projects so leveraged:**
  - Cling/ROOT (C++ interpreter [https://root.cern/cling](https://root.cern/cling))
  - Clang/LLVM (C++ compiler [http://llvm.org](http://llvm.org))
  - PyPy (Python w/ JIT [http://pypy.org](http://pypy.org))
  - CFFI (Python FFI to C [https://cffi.readthedocs.io](https://cffi.readthedocs.io))

<table>
<thead>
<tr>
<th></th>
<th>Lines of C++</th>
<th>Lines of (R)Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPython/cppyy</td>
<td>~18K</td>
<td>~1K</td>
</tr>
<tr>
<td>PyPy/cppyy</td>
<td>~2K</td>
<td>~4K</td>
</tr>
</tbody>
</table>

*Note: we also build this work on an earlier, Reflex-based, version of cppyy. With Cling, we get more functionality, improved ease-of-use, better performance, and C++1x.*
High-performance Python-C++ bindings with PyPy and Cling

Architecture
(example of function calls shown)

Two paths:
7A-10A: wrappers
7B-10B: direct FFI
Both Python and Cling are interactive, allowing:

- Automatic template instantiations
  - Transparent unique_ptr<>, shared_ptr<>, etc.
- std::vector<> optimizations
- Offset calculations for multiple virtual inheritance
- Cross-language derivation (both ways)
- Etc.?! More ideas to explore ...

C++1x much better at expressing ownership

- Improved automatic memory management
- Also targeted semi-manually with pythonizations
  - E.g. name-based, custom smart pointers, etc.
Optimizations

- PyPy JIT is conservative and optimizes Python
  - JIT hints needed to “teach it C++,” e.g.:
    - Class hierarchies are fixed (and so are most offsets)
    - Side-effect free functions are elidable
    - Specialized paths (e.g. lookups, overloading, FFI)
  - Need micro-benches to debug & verify performance
    - Hints are elementary, so scales to more complex codes

- Set of micro-benchmarks follows
  - Not feature-set exhaustive (yet)
  - Comparisons made with:
    - Target: optimized C++
    - CPython/cppyy: the default of most of our users
    - Swig: well-known, widely used
Micro-benchmark: empty function call

Remaining overhead is GIL release/re-acquire (~20x). pypy-c pure inlines.
Micro-benchmark: “complex” function call

Remaining overhead is GIL release/re-acquire (~3x).

High-performance Python-C++ bindings with PyPy and Cling
Swig tries methods in order, cppyy hashes successful calls. FFI suffers from GIL.
Micro-benchmark: data member access

SWIG creates Python properties in Python, CPython/cppyy in C++.

High-performance Python-C++ bindings with PyPy and Cling
Micro-benchmark: \texttt{std::vector<int>}

There's a frame left in FFI path; \texttt{pypy-c pure} uses \texttt{array.array}
Realistic Code

Creates values, applies some math, makes selections, store in histograms and ntuple format, write to disk.
Caveats

- Non-JITed pypy-c is $\sim2x$ slower than CPython
  - Is code generation problem; don't expect fix

- PyPy uses a true garbage collector
  - C++ destructors called “randomly”
    - Can call gc.collect() explicitly to force calls
    - No guarantee that destructors will be called on exit
  - No true RAII possible

- PyPy JIT can be fickle
  - Inner loop branches take a long time to heat up
  - Minor code changes can cause performance drops
Distribution

- Two modules and a pip for externals
  - cppyy in PyPy is builtin
    - Currently on cling-support branch; on main soon
  - cppyy for CPython is extension module
    - In most Linux distros, MacPorts, etc. (as part of ROOT)
  - Pip package with externals (for PyPy) to be released

- Licenses:
  - All open source, all very permissive
Conclusions

- We developed Cling-based Python-C++ bindings
  - Supports C++1x and beyond
  - Supports large C++ codes
  - High performance with PyPy
- Combined interactive C++ with Python
  - New functionality and optimizations
- Showed 3x improvement for realistic code

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