PyROOT: PyCling and Cppyy

Status of the transition and future plans

Wim Lavrijsen (LBNL)

ROOT Users Workshop

March 12, 2013, Saas Fee, Switzerland

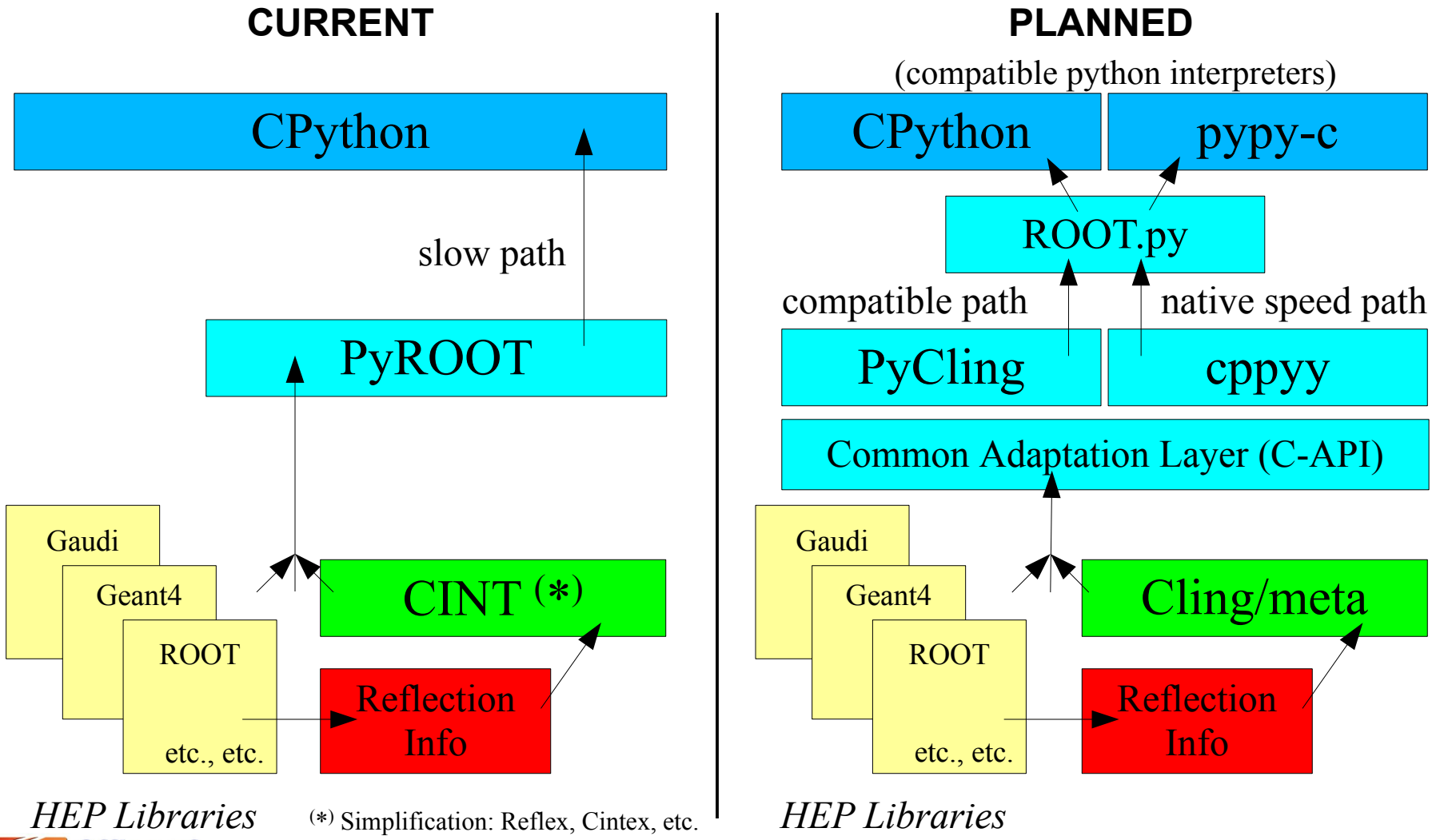
New developments: Motivation

- New and old challenges come to a point
  - Cling in ROOT6 (and removal of CINT/Reflex)
    - And arrival of C++11 that it will support
  - General “slowness” of Python
  - Multi-/many-core CPUs now mainstream
  - Long-standing issues such as standalone usage
- New technology provides opportunities
  - Cling provides a real compiler (CLang/LLVM)
  - PyPy, providing a JIT for Python, has come of age
    - Release 2.0 to come out soon
- Enough critical mass exists to evolve PyROOT

PyROOT on Cling

- Initial version based on enhanced ROOT/meta
  - ROOT/meta set to replace all uses of Reflex API
    - TInterpreter, TClass, TMethodCall, etc.
  - PyROOT is a good use case for debugging meta
    - Uses (and has test cases for) almost every feature
    - Other big users: e.g. ROOT I/O
  - Changes to PyROOT have been relatively limited
    - E.g. removal of G__*; but bulk of the work is in meta and Cling
- Next iteration will provide a split
  - Thin C-API shared with PyPy/Cppyy (see later)
  - PyCling on Cling standalone (not in ROOT)
  - PyROOT on PyCling: adding ROOT pythonizations

Python-bindings architecture



PyROOT on Cling: Status

- Passes about 90% of unit tests; still missing:
  - ROOT/meta: object by-value, template functions
  - PyROOT: callbacks (TF1/2/3, TMinuit, etc.)

  => hope to hit 100% for release v6.00.00
- $ROOTSYS/pyroot/tutorials mostly okay
  - Some remaining issues with graphics (TQxyz callbacks)
- Currently significantly slower
  - E.g. for hsimple.py: startup x10, run-time x40
    - Not counting graphics, which equalizes somewhat
  - Causes: extra layers and inefficient workarounds

  => expect to solve this for release v6.02.00

New features

- Multiple virtual inheritance *fully* supported
  - Uses Cling to generate Reflex-like stubs on-the-fly
- C++11 (to the extent CLang supports it)
  - New C++11 declarations (resolve to simpler terms)
    - E.g. auto → real type after the compiler is done with it
  - C++11 *implementations* not visible to bindings
    => automatically okay
  - Semantics of move constructors is open question
    - Python does not distinguish *l*- from *r*-values
    - Might be useful in case of Python just-in-time compilation
  - C++11 concurrency probably not needed in bindings
    - Python (and ROOT) have their own concurrency classes

C++11 example


```
$ cat cpp11.C
constexpr int data_size() { return 5; }
auto N = data_size();

template<class L, class R> struct MyMath {
   static auto add(L l, R r) -> decltype(l+r)
   {
      return l+r;
   }
};
template class MyMath<int, int>;      // safe!

$ cat cpp11.py
import ROOT
ROOT.gROOT.LoadMacro('cpp11.C')
print 'N =', ROOT.N
print '1+1 =', ROOT.MyMath(int, int).add(1,1)

$ python cpp11.py
N = 5
1+1 = 2
```

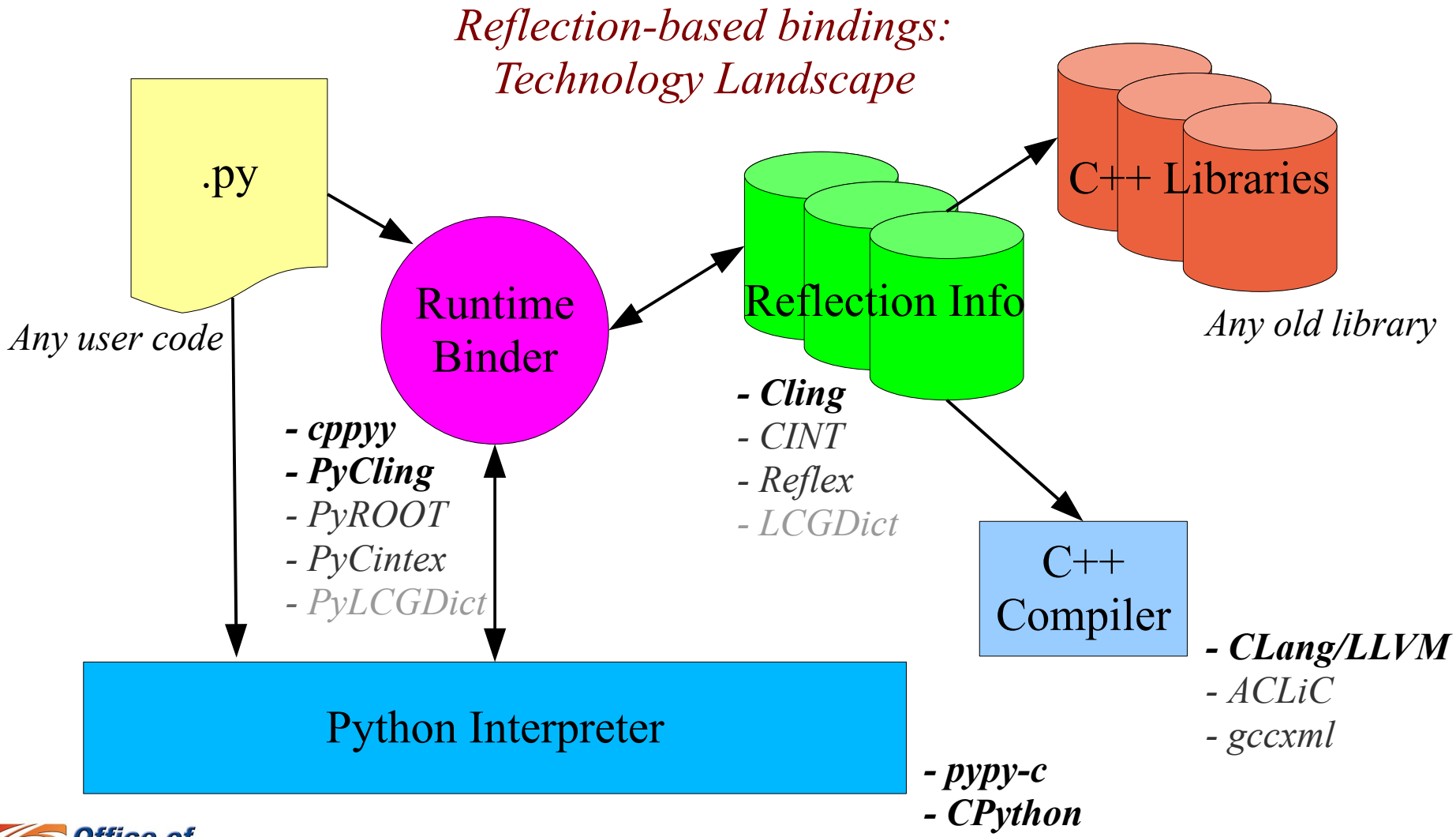
Cppyy: C++ bindings for pypy-c

PyPy and pypy-c

- A dynamic language development framework
  - Framework itself is implemented in (R)Python
  - Provides models for objects, memory, threading, etc.
  - Generates a *tracing JIT* for the dynamic language
- Alternative implementation to CPython: pypy-c
  - Developed within the PyPy framework
    => makes it “Python written in Python”
  - Offers higher computational speeds through the JIT
  - Lowers memory footprint compared to CPython
  - Provides transparent path to multi-core usage through Software Transactional Memory (STM)

See: http://pypy.org/

Python-bindings in HEP



cppyy & CppyyROOT

- cppyy provides PyPy based on C++ reflection
  - Builds on experience from PyROOT & its siblings
  - CppyyROOT.py: ROOT.py compatible
    - To be merged at some point, leaving only ROOT.py
- Works b/c reflection info offers two main features

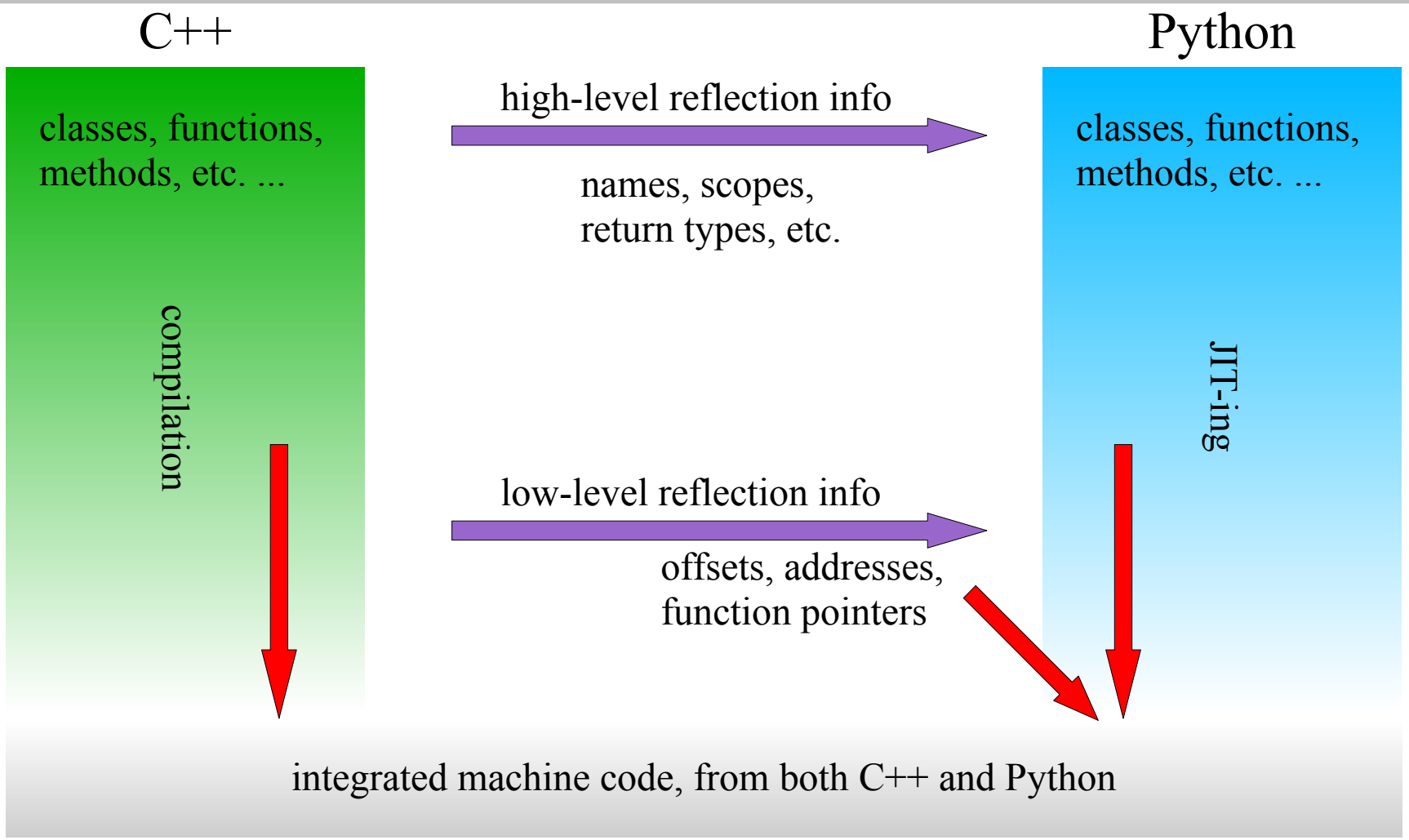
High-level structure
for abstractions and
user representation
(e.g. class names)

=> from CINT, Reflex, Cling

Low-level details
for deconstruction
needed for JIT-ing
(e.g. function ptrs)

=> from Reflex, Cling

cppyy: Un-boxing of C++ code



cppyy: features

- Bulk of language mapping is implemented:
  - Builtin types, pointer and array types
  - Namespaces, global functions, global data
  - Default variables, return object by value
  - Classes, inner classes, static/instance data members, methods
  - Single and multiple inheritance, (mixed) virtual inheritance
  - Templated classes, basic STL support and pythonizations
  - Basic (global) operator mapping
  - Both Reflex and CINT back-ends (latter missing fast path)
- Short-list of important missing features:
  - Memory mgmt heuristics, several C++ corner cases
  - Fast garbage collection for C++ temporaries
  - In CppyyROOT.py: PyROOT-equivalent pythonizations

Unboxing TTrees example (from CHEP'12)

```
// retrieve data for analysis
TFile f = new TFile("data.root");
TTree t = (TTree*)f->Get("events");

// associate variables
Data* d = new Data;
t->SetBranchAddress("data", &d);

Long64_t isum = 0;
Double_t dsum = 0.;

// read and use all data
Long64_t N = t->GetEntriesFast();
for (Long64_t i=0; i<N; i++) {
   t->GetEntry(i);
   isum += d->m_int;
   dsum += d->m_float;
}

// report result
cout << isum << " " << dsum << endl;
```

```
# retrieve data for analysis
input = TFile("data.root")

# read and use all data
isum, dsum = 0, 0.
for event in input.data:
   isum += input.data.m_int
   dsum += input.data.m_float

# report result
print isum, dsum
```

Python allows boilerplate code to be hidden through hooks in the language

Note: simplistic example chosen to make sure that language overhead fully dominates *rather than I/O or object construction.*

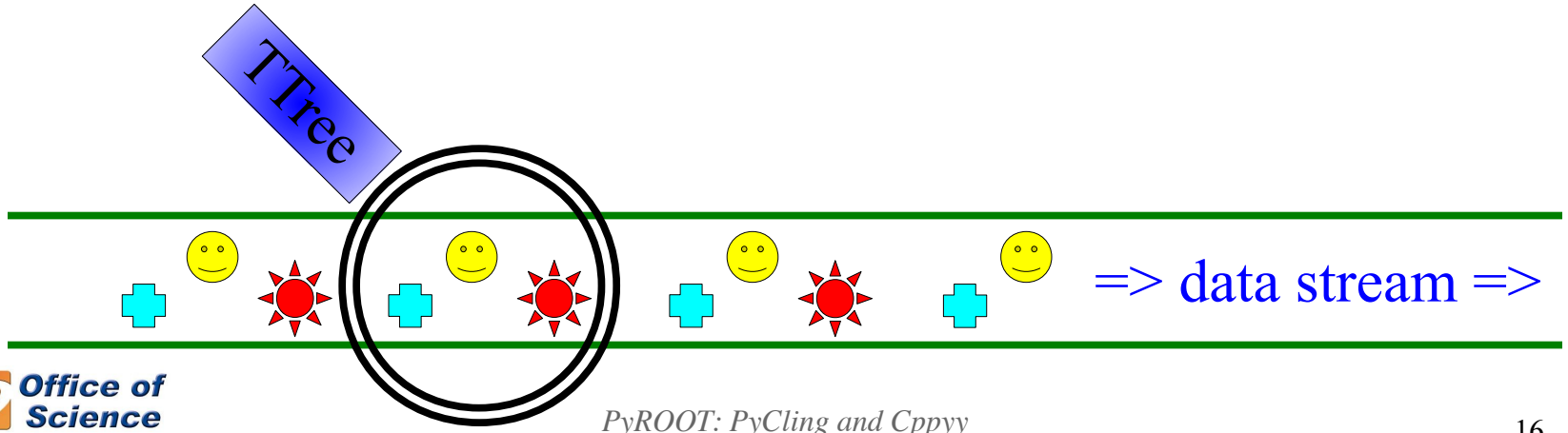
Nice syntax … … but not so nice speed

- Nice syntax causes not so nice slow-down:
  - C++ ..... 100,000,000 “events”: 13.4 secs (1x)
  - Python ..... 100,000,000 “events”: 640.2 secs (50x)
- Cause: language hooks are of a general nature
  - Hooks go from Python, through C++, and back
    - Results in several call layers and lots of temporary objects
  - In comparison, C++ language overhead is *zero*
    - Data members in struct object are accessed directly
- Could the lost performance be regained?
  - *While keeping the nice syntax intact?*
  - Can the inter-language layering be removed?

TTree == "dispersed TClass"

- TTrees represent memory layouts
  - Like TClasses, except dynamically setup/collected
  - Boilerplate code establishes the connections
- TTree is conceptually a "focusing lens":
  - Once memory layout is established, it is mostly static
  - *Conceptually*, data stream "moves underneath"

=> Use TTree info like TClass info as done in cppyy



Example results *UPDATED!*

- Original results:
  - C++ ..... 100,000,000 “events”: 13.4 secs (1x)
  - Python ..... 100,000,000 “events”: 640.2 secs (50x)
- Exact same Python code, but now JIT-ed TTree:
  - PyPy ……. 100,000,000 “events”: 14.2 secs (1.06x)

Note 1: at CHEP'12, was 2.7x, and used 10E7 “events”
Note 2: cppyy startup is 0.32s more than root.exe

- Closer to C++ w/ more code in loop or if I/O bound
  - But: data classes are slower on CINT (no function ptrs)
- Selective reading: only read branches actually used
  - Allows JIT-ed code to improve on *naïve* C++

Python with multicore

- CPython hampered by the *Global Interpreter Lock*
  - No real CPU-intensive parallelism possible on threads
    - Workaround: multiprocessing, with thread-like interface
  - Fine for mixing e.g. CPU and I/O parallelism
- PyPy: use software transactional memory (STM)
  - Build communicating sequential programs from traces
    - Automatically, or with limited user hints (think: OpenMP)
  - First prototype for PyPy delivered
    - But no JIT support yet, and suffers from resource leaks
    - Slow: currently, need at least 4 cores to break even
  - Provides *uni-processor* view of many-core system
  - Limited hardware support for TM on Intel's Haswell

Automatic parallelization

- Whole-program parallelization with PyPy and STM
  - Expected to get you within 2x of optimal performance
    - May sound bad, but it's free, and should scale well
  - Works well with added hand-written concurrency
    - I.e. can still do further optimization *after* the code works
- Idea: add HEP-specific software patterns to JIT
  - Similar to what was done for TTrees
  - Hand JIT transactions on a platter for better scheduling
  - Hand JIT data model for auto-vectorization
    - Vectorization is often more important than threading

=> Get close to optimal, *out-of-the-box*, for typical HEP codes

Pythonic ROOT

- PyROOT is mostly about automatic bindings
  - Sine qua non: unwieldy and unmaintainable otherwise
  - Dictionaries (incl. for experiment data) widely available
    - And maintained for I/O and CINT in experiment releases
- Automatic bindings often feel too much like C++
  - Some limited, still generic, Pythonizations exist
    - E.g. TFn, TTree, looping over std::vector, etc.
- Solution: rootpy provides a more pythonic ROOT
  - See Noel's talk on Wednesday afternoon

Summary and planning

- PyROOT on Cling set for availability in v6.00.00
  - Final todo-list: C++ objects by-value and callbacks
- PyCling scheduled for v6.02.00
  - Work on refactoring and speed-up
  - Use C-API designed for cppyy
  - More features (e.g. cross-language inheritance)
- cppyy available in PyPy repository and on AFS
  - CINT and Reflex back-ends, Cling back-end planned
  - Can now read from TTrees at optimized C++ speeds
  - http://doc.pypy.org/en/latest/cppyy.html
  - /afs/.cern.ch/sw/lcg/external/pypy/x86_64-slc5

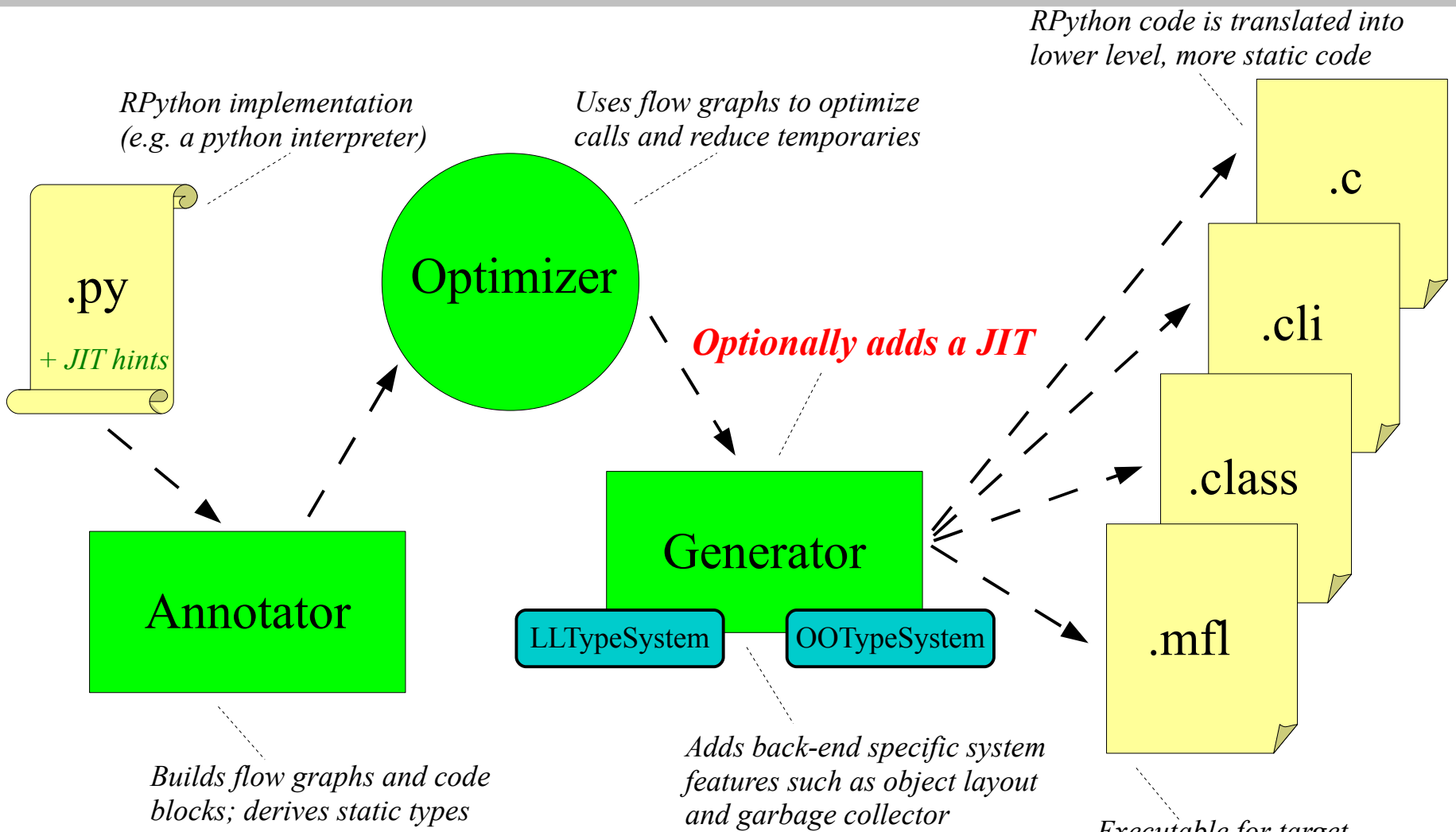
That's All Folks!

(back-up slides follow)

Backup slides:

- PyPy tool chain
- Tracing JIT
- Traces, guards, branches
- Prototype performance (Reflex)
- Prototype performance (2, Reflex)

PyPy tool chain



PyPy's generated JIT

- JIT applied at the interpreter level
  - Optimizes the generated interpreter for a given input
    - Where input is the user source code and application data
  - Combines light-weight profiling and tracing JIT
    - Especially effective for algorithmic, loopy code
- Can add core features at interpreter level
  - Interpreter developer can provide hints to the JIT
    - Through JIT API in RPython
    - Elidable functions, promotable variables, libffi types, etc.
  - JIT developer deals with platform details
  - All is completely transparent for end-user

"Classic" v.s. Tracing JIT

- "Classic" just-in-time compilation (JIT):
  - Run-time equivalent of the well-known static process
    - Profile analysis to find often executed ("hot") methods
    - Compile hot methods to native code
  - Typical application for interpreted codes
- Tracing just-in-time compilation:
  - Run-time procedure on actual execution
    - Locate often executed hot paths (e.g. loops)
    - Collect linear trace of one path (e.g. one loop iteration)
    - Optimize that linear trace
    - Compile to native if applicable
  - Can be used both for binary and interpreted codes

Tracing JIT

Program code:

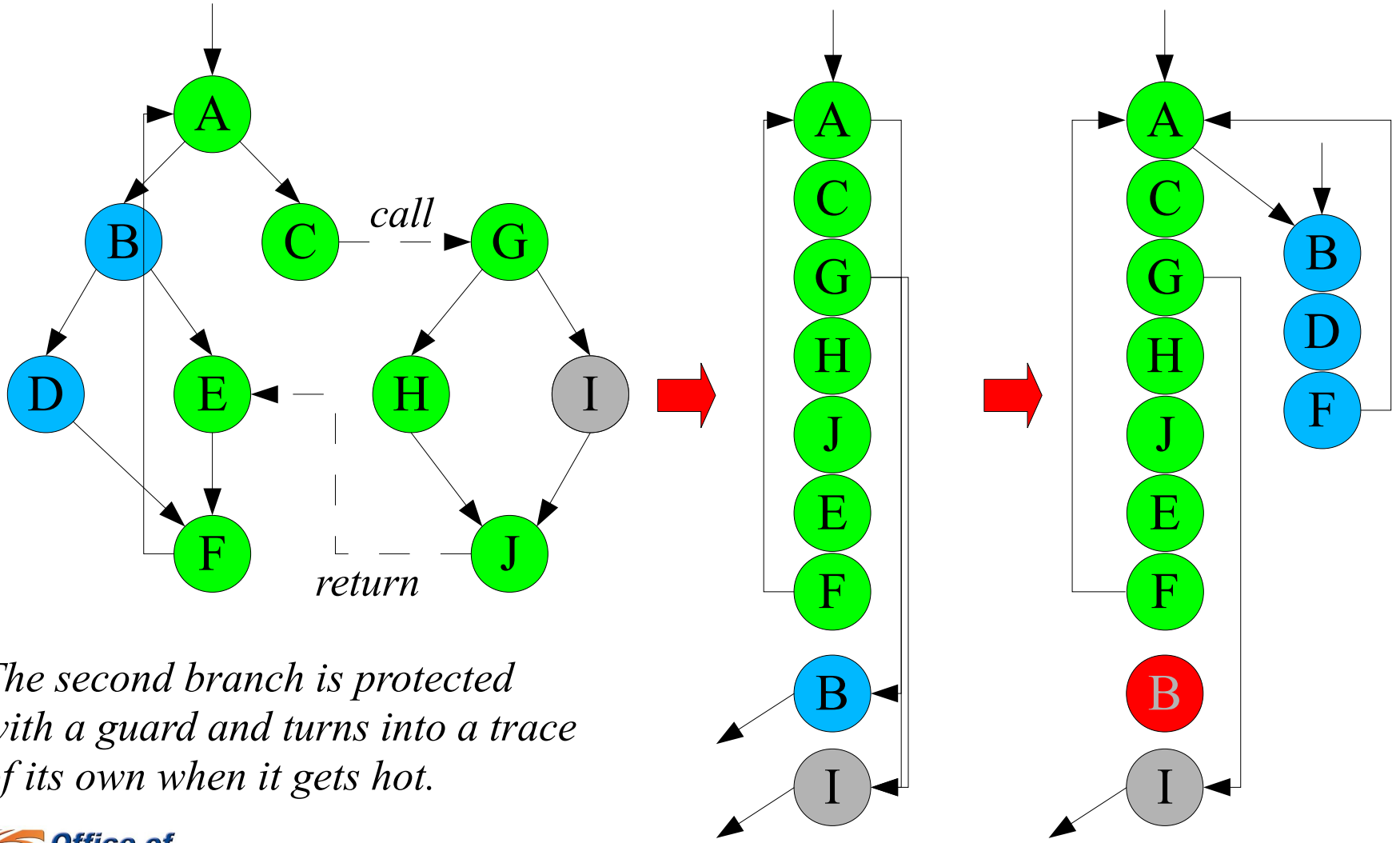
```
    A:
L:  cmp
    inst_a1
    inst_a2
    jne aa          →     call C:
    Call            →     B:
                          inst_b1
                    ←     return
    inst_aN
    goto A
```

Linear trace:

inst_a1, inst_a2, G(aa), inst_b1, inst_aN

- In interpreted mode:
  - Process user code
  - Identify backwards jumps
  - Collect trip counts
- If threshold crossed:
  - Collect linear trace
  - Inject guards for all decision points
  - Optimize trace
  - Compile trace
  - Cache & execute
- In compiled mode:
  - Process user code
  - Collect trip counts on guards
- If threshold crossed for guards:
  - Create secondary trace
  - Rinse & repeat

Traces, guards, branches



The second branch is protected with a guard and turns into a trace of its own when it gets hot.

Prototype performance
(on Reflex)

- Benchmark measuring bindings **overhead only**:
  - SWIG: 7.3 (500x)
  - PyROOT: 4.7 (300x)
  - pypy-c-jit: 0.70 (50x)
  - pypy-c-jit-fp: 0.063 (4x)
  - pypy-c-jit-fp-py: 0.125 (8x)
  - C++: 0.015 (1x)

Notes:
1) *"overhead" is the price to pay when calling a C++ function that is overloaded on different types*
2) *bindings overhead matters less the larger the C++ function body*
3) *"-fp" is "fast path" and requires Reflex patch*
4) *"-py" is the pythonified (made python-looking) version, which still needs to be made JIT-friendly*
5) *"C++" is gcc -O2 (other codes are -O2 as well)*

Prototype performance (2)
(on Reflex)

- Overhead w/ "realistic" C++ function body:
  - SWIG: 7.5 (28x)
  - PyROOT: 5.0 (20x)
  - pypy-c-jit: 0.85 (3x)
  - pypy-c-jit-fp: 0.27 (1x)
  - pypy-c-jit-fp-py: 0.28 (1x)
  - C++: 0.27 (1x)

Notes:
1) *"Realistic" means "a lot" of computation being done in the C++ function body: the atan() function is called*
2) *"-fp" is "fast path" and requires Reflex patch*
3) *"-py" is the pythonified (made python-looking) version, which still needs to be made more JIT-friendly*
4) *"C++" is gcc -O2 (other codes are -O2 as well)*
5) *Note the benefit of OOO: numbers are not straight sums (they are on a P4; this is benched on a SandyBridge)!*