CRD

C O M P U T A T I O N A L   R E S E A R C H   D I V I S I O N

BERKELEY LAB

Optimizing Python-based ROOT I/O

With PyPy's Tracing Just-In-Time Compiler

Wim Lavrijsen (LBNL)

Computing in High Energy and Nuclear Physics 2012
May 21-25, New York University, New York

Presented by: Roberto Vitillo (LBNL)

Office of Science
U.S. DEPARTMENT OF ENERGY

Python: nicer syntax ...


```
// 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 << sumi << " " << sumd << 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

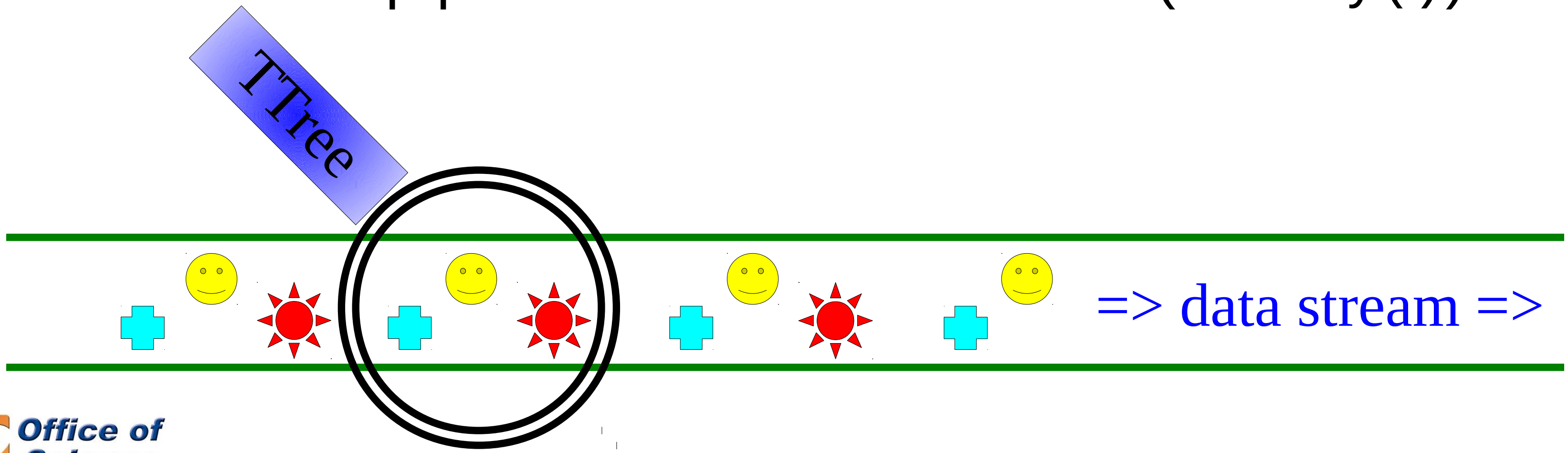
Note: simplistic example chosen to make sure that language overhead fully dominates *rather than I/O or object construction.*

… but not so nice speed

- Nice syntax causes not so nice slow-down:
  - C++ .... 10,000,000 “events”: 1.26 secs
  - Python ..... 10,000,000 “events”: 68.7 secs (55x)
- Cause: language hooks have 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?
  - Can Python learn “natively” about TTrees?

TTree == “dispersed TClass”

- TTrees represent memory layouts
  - Like TClasses, except dynamically setup/collected
  - Boilerplate code establishes the connections
- TTree is a “focusing lens”:
  - Once memory layout is established, it is mostly static
  - *Conceptually*, data stream “moves underneath”
  - New setup possible for next file/chain (`Notify()`)



CRD

“TClass” already solved: PyPy/cppyy

BERKELEY LAB

- Utilizes a *tracing just-in-time compiler*:
  - Remove layers by inlining or eliding function calls
  - Resolve temporaries through escape analysis
    - Morphed into stack objects or resolved completely
  - Promote constants through invariant code motion
- Utilizes *C++ reflection info*:
  - Build up nice pythonistic representations
  - Break down calls and data access to memory pointers
    - Subsequently injected into JIT-generated machine code
    - Final, integral result runs at native speeds

=> Same techniques can be applied to TTrees!

"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

CRD

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:

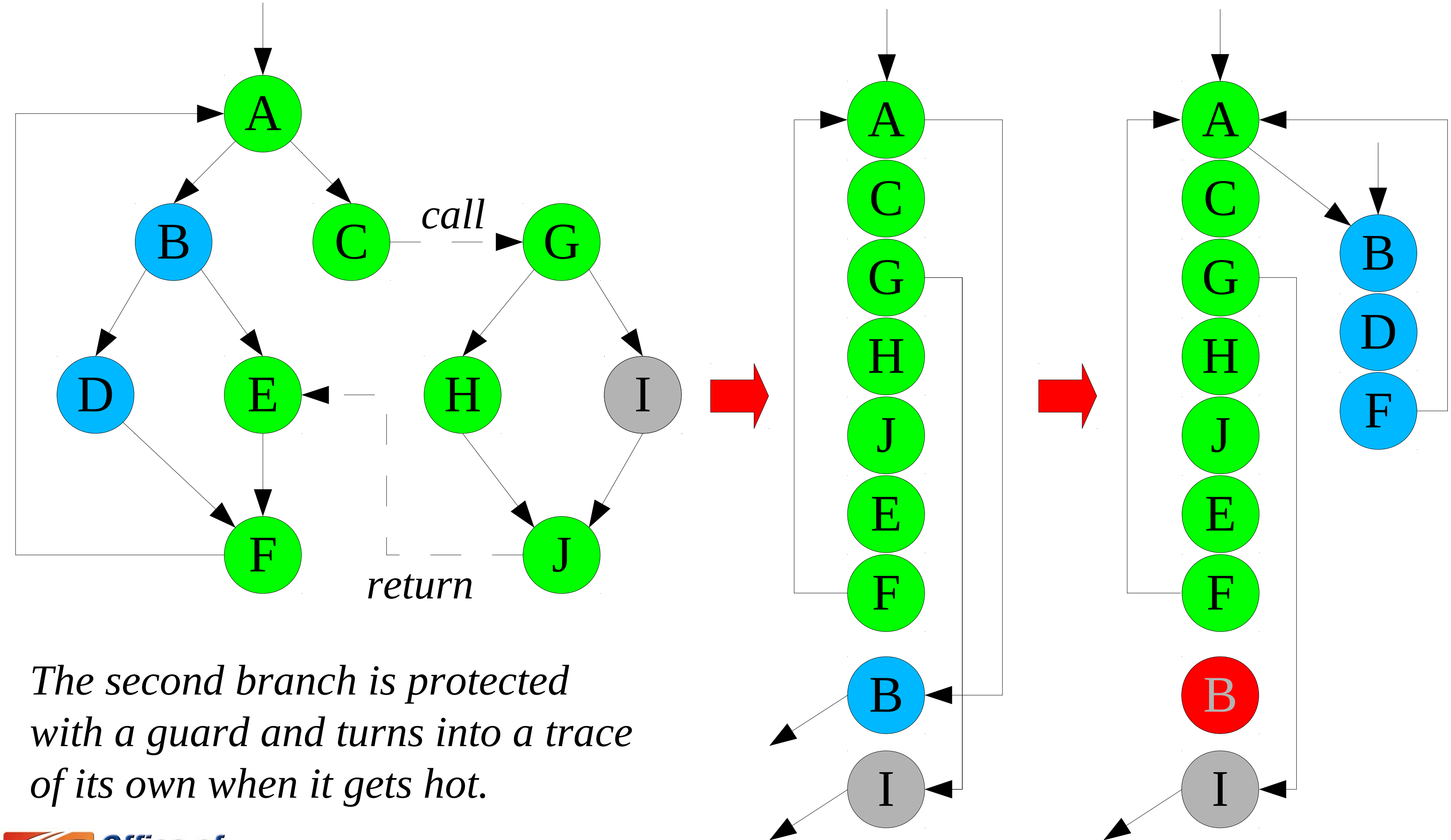
inst_a1, inst_a2, G(aa), inst_b1, inst_aN

- 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

CRD

Traces, guards, branches

BERKELEY LAB

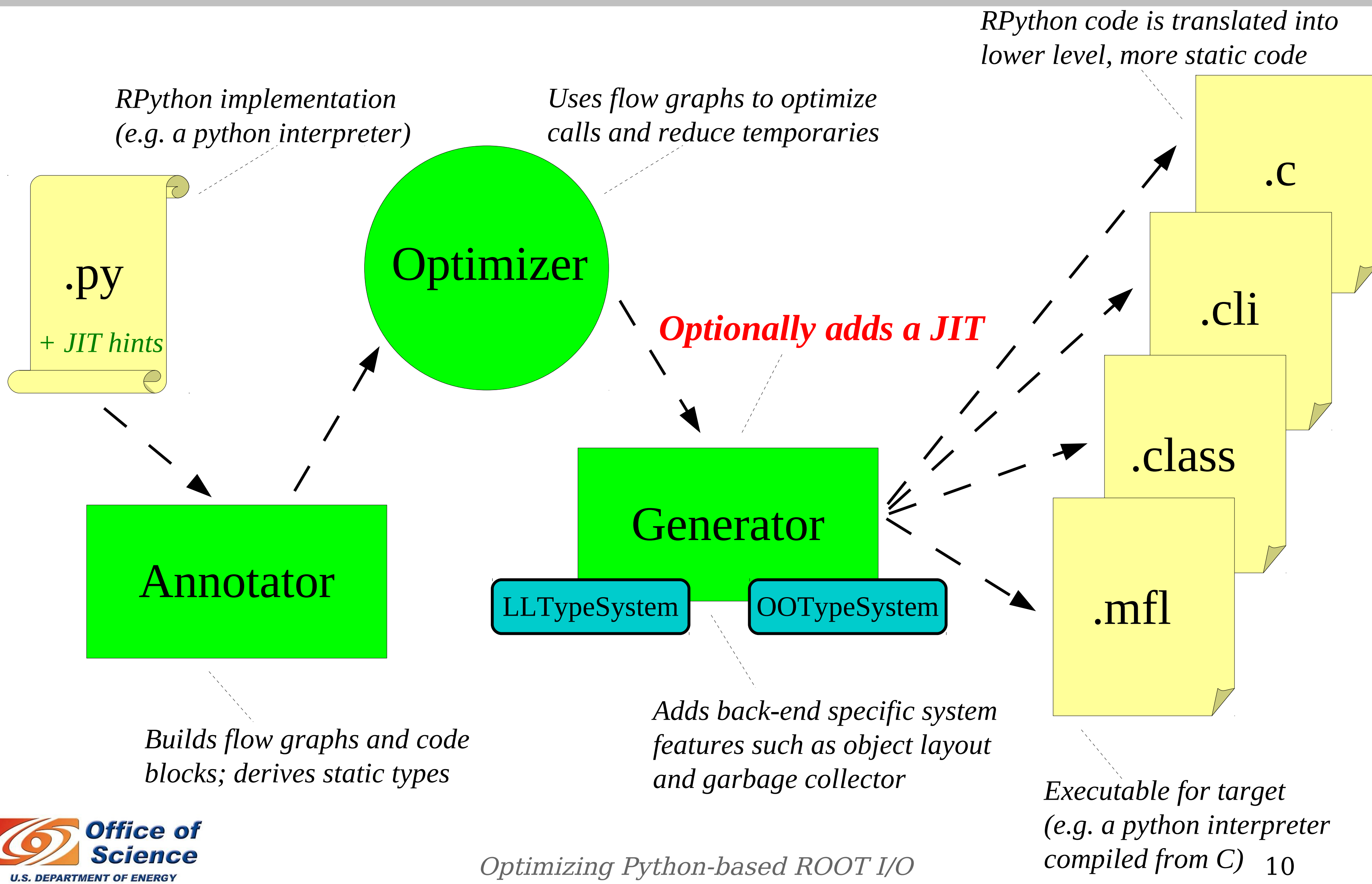


The second branch is protected with a guard and turns into a trace of its own when it gets hot.

What is PyPy?

- A dynamic language development framework
  - Framework itself is implemented in (R)Python
  - One language/interpreter thus developed is Python
    - Most advanced of the languages developed in PyPy
    - An alternative implementation to CPython
    - Makes it “Python written in Python” as it is best known for
- A translation tool-chain with several back-ends
  - Adds object, memory, threading, etc. models
  - E.g. RPython => C to get **`pypy-c`** (compiled)
- A tracing JIT generator as part of the toolchain
  - Operates on the *interpreter* level (hence: “meta-JIT”)

PyPy Toolchain



PyPy's generated JIT

- JIT applied on 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

cppy

- Builds PyPy bindings from C++ reflection
  - Lots of experience from PyROOT & its siblings
  - Compatible version being developed: CppyyROOT
- Reflection info offers two main features:

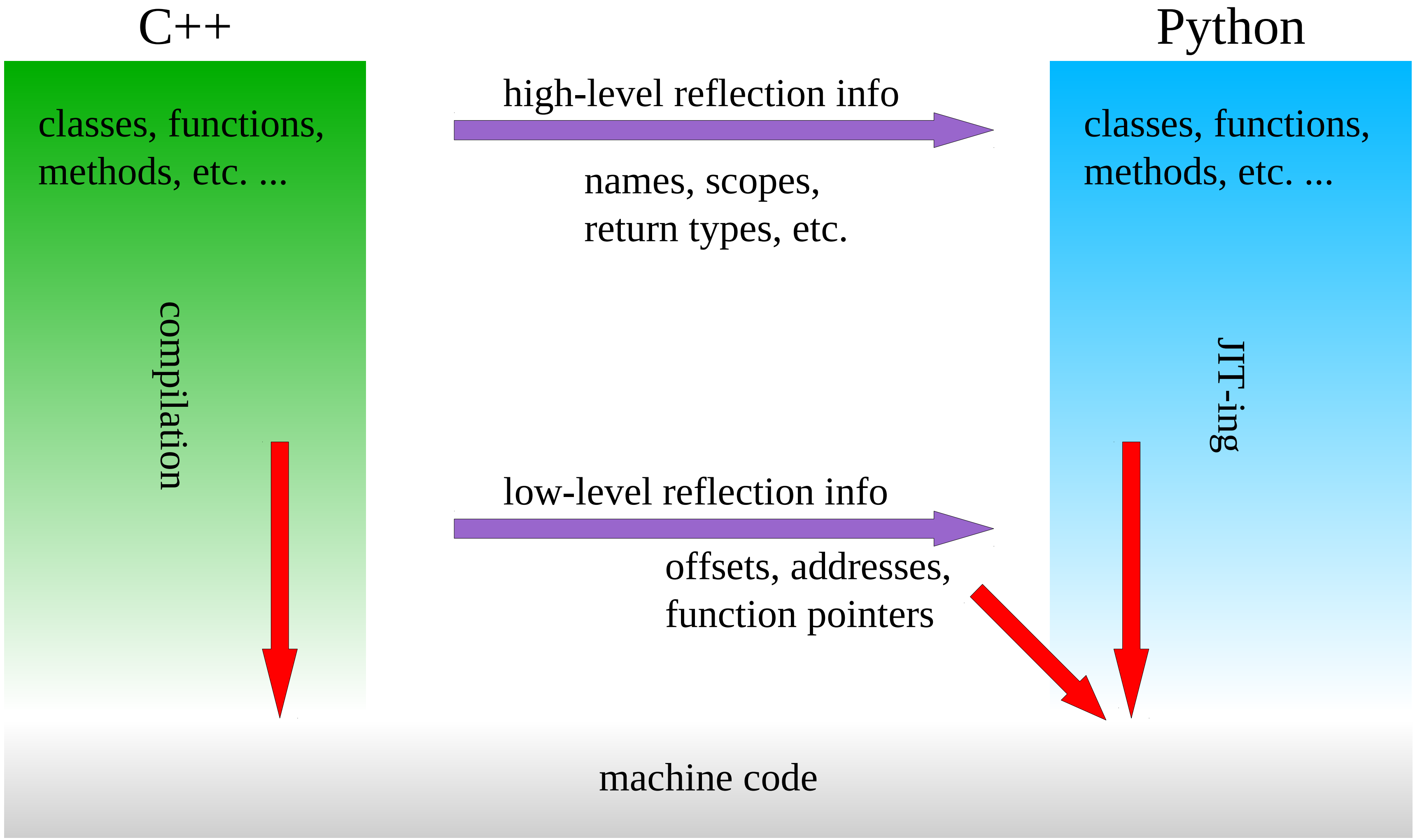
High-level structure for abstractions and user representation

Low-level details for deconstruction needed for JIT-ing

- Allows break-down to machine-level operations
  - E.g. walks vtables, calculates class offsets, etc.
  - Meets JIT on its own terms, instead of through an API

CRD

Abstractions breakdown



cppyy: currently supported features

- Bulk of C++ -- Python 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 and user control
  - Cling/LLVM precompiled modules back-end
  - Various corner cases (e.g. fast-path C++ exception handling)

New TTree representation, using cppyy techniques


```
$ pypy-c
>>>> import CppyyROOT as ROOT
>>>> input = ROOT.TFile("data.root")
>>>> data = input.data
>>>> print type(data)
<class '__main__.TTree'>
>>>> print data.__dict__
{}
>>>> for event in data:
....     # do analysis
....
>>>> print type(data)
<class '__main__.TTree'>
>>>> print data.__dict__
{ '_pythonized': True,
  'data': <__main__.Data object at 0x00007f99407a1be0>}
>>>>
```

Automatically generated based on branch list and branch class names

=> *TTree representation constructed on and managed per instance to prevent life-time issues and allow TTrees to be still typed as TTree*

CRD

JIT-ed TTree performance

BERKELEY LAB

- Original results:
  - C++ .... 10,000,000 “events”: 1.26 secs (1x)
  - Python ..... 10,000,000 “events”: 68.7 secs (55x)
- Exact same Python code, but now JIT-ed TTree:
  - PyPy ……. 10,000,000 “events”: 3.45 secs (2.7x)
- Not (yet) 1x, b/c of guards (C++ is direct access)
  - Need guards removal by allowing JIT to freeze TTrees
- Closer to C++ w/ more code in loop or if I/O bound
  - Data classes with a default constructor or T/P separation
    - May even require more CPU-intensive decompression
  - Selective reading (more work/CPU for buffering scheme)

Office of Science
U.S. DEPARTMENT OF ENERGY

CRD

Conclusions & Plans

Berkeley Lab

Huge improvement in Python-based ROOT I/O has been achieved using PyPy's tracing JIT!

- Laundry list of TODO items:
  - Further improvement by freezing TTree outside loop
    - Get away with fewer guards on data member access
    - With out-of-order execution, 1x should be possible
  - Make CppyyROOT fully PyROOT compatible
    - In particular, resolve casts needed for TTree writing
  - Automatic (de)activation of branches on use in traces

Resources

- Code repository (PyPy):
  - https://bitbucket.org/pypy/pypy
  - Branch: “reflex-support” (soon to move to “default”)
- Documentation for PyPy/cppyy:
  - http://doc.pypy.org/
  - http://doc.pypy.org/en/latest/cppyy.html
- CppyyROOT and CERN installations (ATLAS):
  - http://twiki.cern.ch/twiki/bin/view/AtlasProtected/PyPyCppyy
  - /afs/.cern.ch/sw/lcg/external/pypy/x86_64-slc5

CRD

That's All Folks!

Backup slides:

- List of existing tracing JITs
- Dynamo for PA-RISC
- Benefits of tracing JITs
- Reflection based Python bindings
- cppyy performance

Examples of Tracing JITs

- Dynamo for PA-RISC binary
- PyPy's meta-JIT for Python
- MS's SPUR for Common Intermediate Language
- Mozilla's TraceMonkey for JavaScript
- Adobe's Tamarin for Flash
- Dalvik JIT for Android
- HotpathVM for Java
- LuaJIT for Lua

Dynamo (PA-RISC)

- Of interest because it's a tracing JIT on binary
  - User-mode and on existing binaries and hardware
    - No recompilation or instrumentation of binaries
  - Run-time optimization of *native* instruction stream
- Gained over static compilation because:
  - Conservative choice of production target platforms
    - Incl. legacy binaries existing on end-user systems
  - Constraints of shared library boundaries
    - Actual component to run only known at dynamic link time
    - Calls across DLLs are expensive
  - Linear traces simpler to optimize than call graphs

Tracing JIT Optimizations

- To the linear trace itself, e.g. for guards:
  - Removed if implied, strenghten for larger role
- Loop unrolling and function inlining
- Constant folding and variable promotion
  - Much more effective at run-time than statically
- Life-time and escape analysis:
  - Move invariant code out of the loop
  - Place heap-objects on the stack
- Load/store optimizations after address analysis
  - Collapse reads, delay writes, remove if overwritten
- Parallel dynamic compilation

Benefits of Tracing JIT (1)

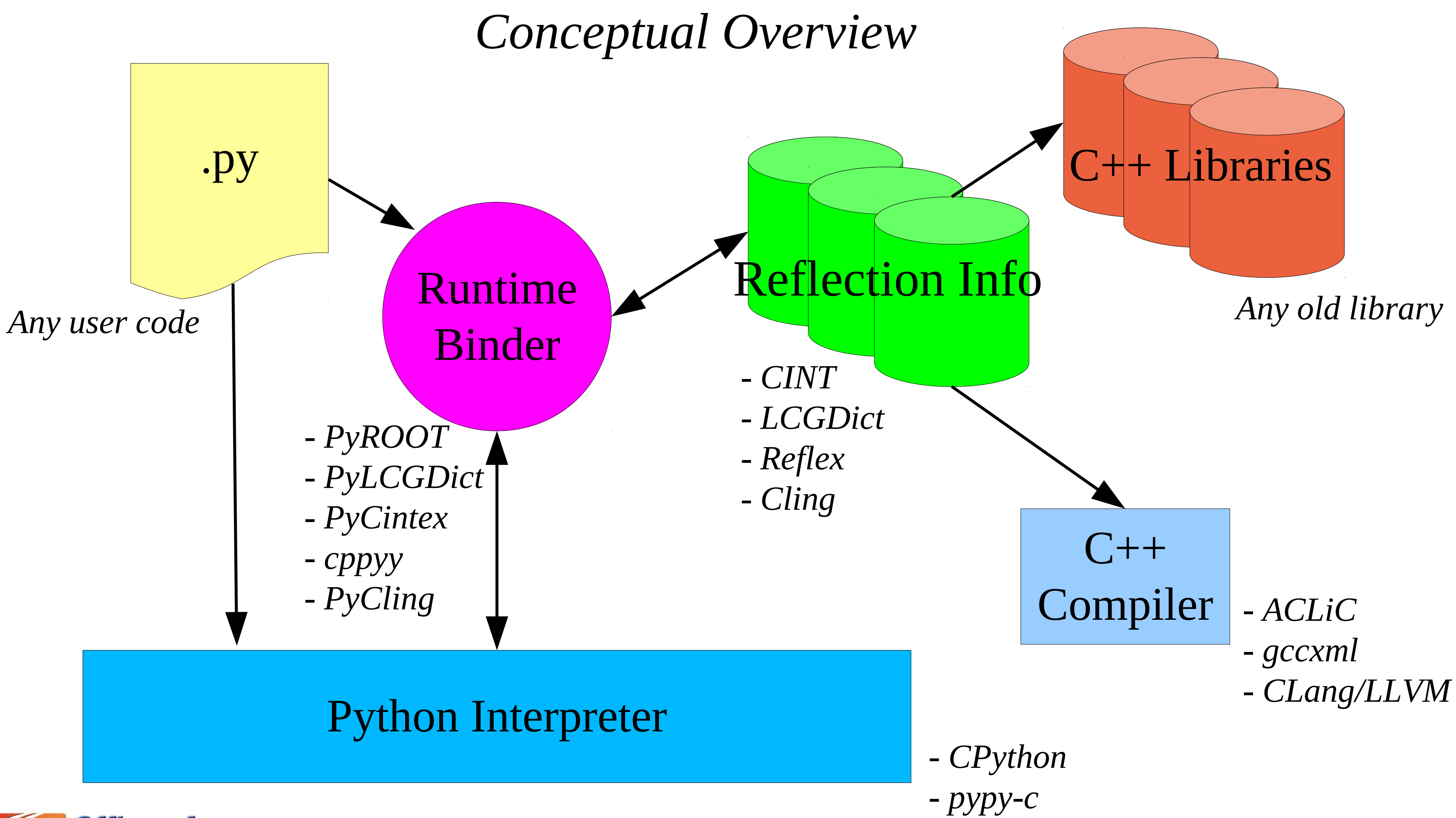
- Profile on current and actual input data on hand
  - ATLAS: huge variety in shape of physics events
- Compile to actual machine features
  - HEP: restricted by oldest machines on the GRID
- Inline function calls based on size and actual use
  - ATLAS: many small functions w/ large call overhead
- Co-locate (copies of) functions in memory
  - HEP: huge spread across many shared libraries
- Remove cross-shared library trampolines
  - HEP: all symbols exported always across all DLLs

CRD

Benefits of Tracing JIT (2)

- Remove unnecessary new/delete pairs
  - ATLAS: tracking code copies for physics results safety
- Judicious caching of computation results
  - HEP: predefined by type, e.g. Carthesian v.s. Polar
- Memory v.s. CPU trade-off based on usage
  - HEP: predefined by type (ptr & malloc overhead)
- Smaller footprint comp. to highly optimized code
  - ATLAS: maybe relevant, probably not
- Low-latency for execution of downloaded code
  - ATLAS: not particularly relevant

Reflection-based Python-C++ Bindings



cppy: call performance

- Benchmark measuring bindings **overhead only**:
  - SWIG: 7.3 (500x)
  - PyROOT: 4.7 (300x)
  - pypy-c-cint: 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 an **empty** 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 (patched) Reflex
4) "-py" is the pythonified (made python-looking) version, which still needs to be made somewhat more JIT-friendly
5) "C++" is g++ -O2 (other codes also -O2), on Sandybridge

cppyy: call performance

- Overhead w/ “realistic” C++ function body:

– SWIG:	7.5	(28x)
– PyROOT:	5.0	(20x)
– pypy-c-cint:	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 some computation being done in the C++ function body: here, the atan() function is called

=> OOO makes overhead virtually zero in fast path

2) “-fp” is “fast path” and requires (patched) Reflex

3) “-py” is the pythonified (made python-looking) version

4) “C++” is g++ -O2 (other codes also -O2), on Sandybridge