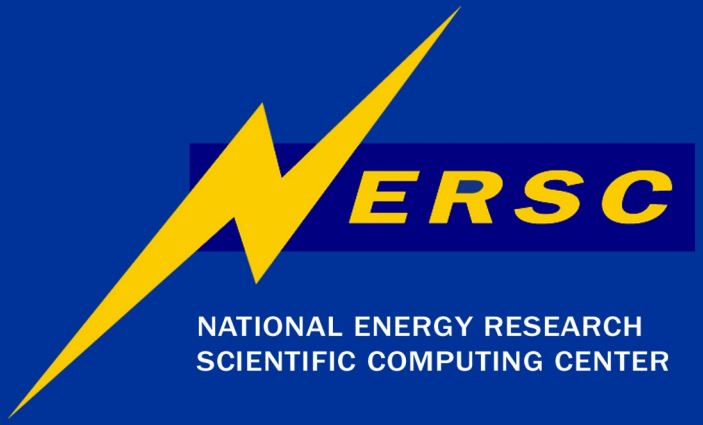


National Energy Research Scientific Computing Center (NERSC)

Optimizations in Python-based HEP Analysis

Sebastien Binet, Wim Lavrijsen

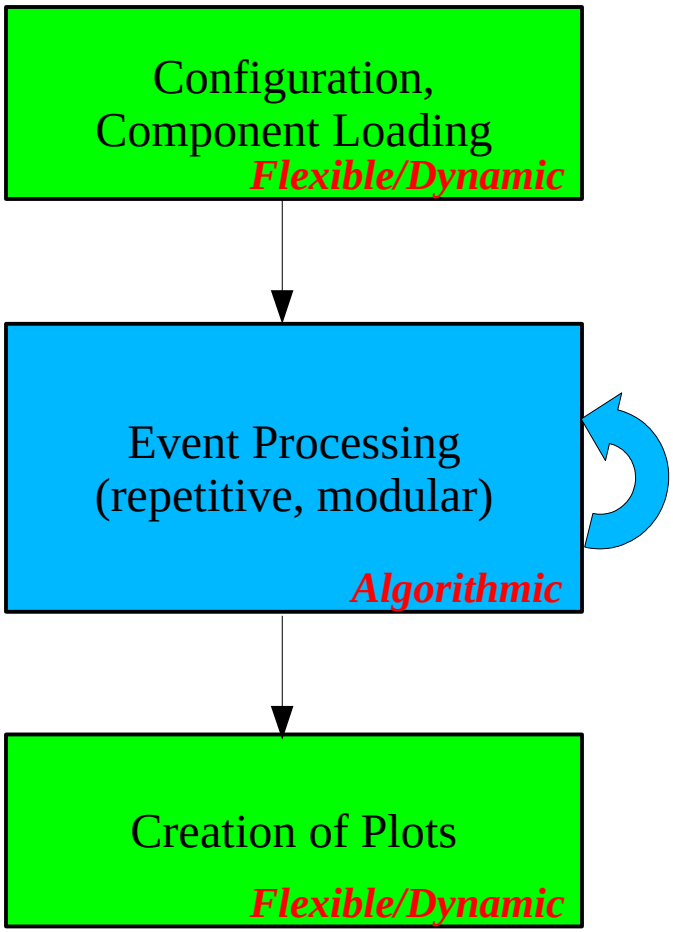
CHEP '07 – Victoria, September 2007

Outline

- **Introduction**
  - Target areas for optimization
- **Code Analyzers and Generators**
  - Psyco, ShedSkin, PyPy
- **Prototyping Results**
  - Optimization of PyROOT-based code
- **Conclusions**
  - Future plans
  - Resources

Introduction

Typical HEP Analysis



- user interface
- string manipulation
- file/shell access

- for loops
- cuts and predicates
- ROOT calls
- math calls
- array, vectors, matrices

=> *target for optimizations*

- selection, ordering, etc. of graphs
- string manipulation
- file/shell access

What to Optimize?

- **Flexibility is great, if it is needed**
  - W/o language support *you* have to write it!
    - You're unlikely to do better than the language
- **Flexibility is a burden, if unneeded**
  - You would pay for what you don't use
    - C++: many complex trade-offs possible
    - Python: typically one, straightforward way
- **In inner loops / algorithmic code:**
  - No flexibility needed
  - Constrained code blocks

=> Optimization

Mock Example

```
>>> from ROOT import gRandom, TCanvas, TH1F
>>> c1 = TCanvas('c1','Example',200,10,700,500)
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> for i in xrange(25000):
...     px = gRandom.Gaus()
...     hpx.Fill(px)
...
>>> hpx.Draw()
>>> c1.Update()
```

} => **inner loop of algorithmic code**

- **Normal, straightforward Python code**
  - **A.k.a. "Fortran-style" Python**
  - **With (readable) tricks: could gain ~20%**

Note: example chosen for clarity, real code uses TH1F::FillRandom('gaus', 25000)

Issues

```
>>> from ROOT import gRandom, TCanvas, TH1F
>>> c1 = TCanvas('c1','Example',200,10,700,500)
>>> hpx = TH1F('hpx','px',100,-4,4)
>>> for i in xrange(25000):
...     px = gRandom.Gaus()
...     hpx.Fill(px)
...
>>> hpx.Draw()
>>> c1.Update()
```

Iterator temp

Method lookups

Object temporary

Bound method temporary

} => *dynamic* **body**

=> Python creates many temporaries

=> Types are considered dynamic

=> No block-level optimizations applied

Code Analyzers and Generators

Available Tools

- **"Integrators"**
  - Pyrex, weave, PyROOT (using ACLiC), etc.
  - C/C++ code (text) blocks in Python with easy sharing of variables
  - Always visible to the end-user
- **"Compilers"**
  - Psyco, ShedSkin, PyPy, etc.
  - Analyze code for algorithmic blocks; translate to lower level language
  - Fully automatable

PyCompilers and HEP code

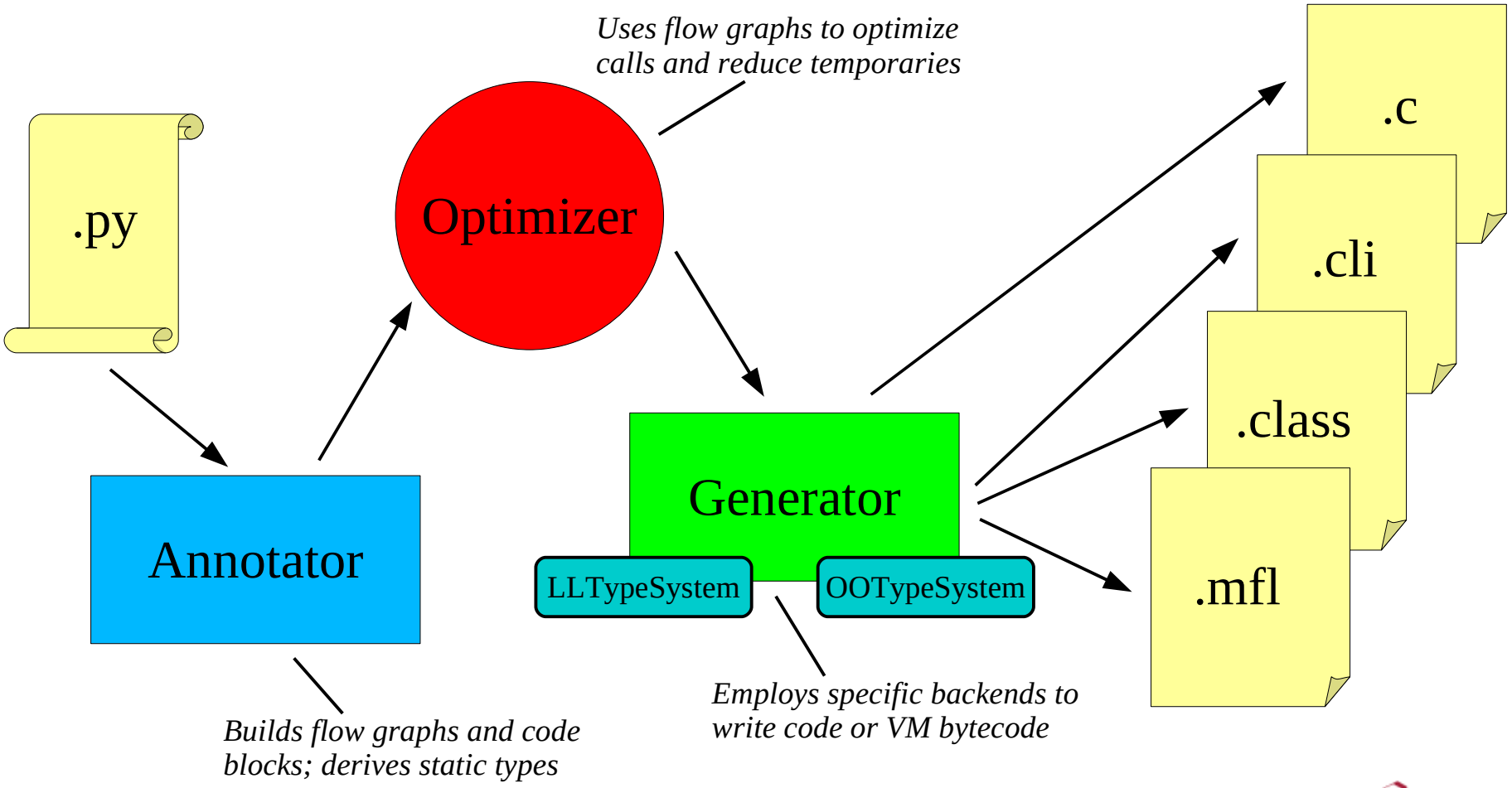
- "Compiler" only handles pure Python
  - Often does include standard libraries
    - Thus, math is usually covered
  - Notably, extension modules are opaque
- HEP code makes use of bindings
  - Quite commonly dictionary based
  - I.e. large fraction of code not handled
    - And hence not optimized ...

=> Idea: feed dict info into compiler

Tool of Choice: PyPy

- **Python interpreter written in Python**
  - Makes Python code first-class objects
    - Allows for analysis and manipulation of code, including the full interpreter
- **Mature and strong development support**
  - Psyco developer is one of the main authors
  - EU "Information Society" project
- **Translation framework**
  - Extensible with external types
  - Fully customizable with new back-ends

PyPy Translation Framework



Annotation

- **Start with known or given types**
  - Constants and built-ins (known)
  - Function arguments (given [when called])
- **Calculate flow graphs**
  - Locate joint points
  - Divide code in blocks

} => graph structure of possible flows and outcomes

- **Fill in all known types**
  - Derived from initial known/given ones
  - Add information from dictionary

Annotation cont.

- **Type deduction need not be complete**
  - Python (e.g. through C-API) as fallback
  - Still allows partial translation/optimization
- **Annotation process fully automatable**
  - Nominally at entry of function call
    - Start (argument) types encountered before?
      - If yes: dispatch to existing translated version
      - If no: build new translated version
  - Can be forced / pre-built semi-manually
    - E.g. by parsing sample code

Annotation Example

user

```
>>> def doFillMyHisto( h, val ):
...     x = ROOT.gRandom.Gaus() * val
...     return h.Fill( x )
...
```

behind the scenes

```
>>> t = Translation( doFillMyHisto )
>>> t.annotate( [ TH1F, int ] )
   -- type(h) is TH1F and type(val) is int
   -- type(x) is float, because:
        Gaus is TRandom::Gaus() which yields (C++)double
        mul( (python)float, int ) yields (python)float
   -- result is None, because:
        Fill is TH1F::Fill which yields (C++)void
>>> doFillMyHisto = t.compile_c()
```

user

```
>>> h, val = TH1F('hpx','px',100,-4,4), 10
>>> doFillMyHisto( h, val )              # normal call
```

Explicitly in translation or at runtime; different (non-)choices can coexist

Optimizations

- **Function inlining**
  - Equivalent to its C++ brother
  - May allow further optimizations
- **Malloc removal**
  - Use values rather than objects
    - Esp. for loop variables and iterators
  - Dict special case: remove method objects
- **Escape analysis and stack allocation**
  - Use stack for scope-lifetime objects

Generation

- **Two type systems**
  - "Low Level" and "Object Oriented"

```
result = method( self, *args )
result = self.method( *args )
```

- **Two kinds of back-ends**
  - Code generation (e.g. C, JS, Lisp)
  - JVM bytecodes (e.g. CLI/.Net and Java)
    - With to-be-compiled code for boundary cases
- **Customizable with hand-written helpers**
  - Cover cross-language calls etc.
  - Automatic based on dictionary info

Prototyping Results

Prototype

- **PyPy's extfunc & type support limited**
  - Primarily targets POSIX functions in C
  - Type support not fully implemented

  => Perform type registration from dictionary

  => Minor logic changes *inside* PyPy annotator
- **Some changes in PyROOT needed**
  - Missing conventional functional variables
    - Used by PyPy for descriptive purposes
  - Currently handled by modding PyPy instead

Back-end Cheat

- **Use helper funcs to call C++ from C**

```
extern "C"
  double TRandomGaus( void* self ) {
     return ((TRandom*)self)->Gaus();
  }
  void TH1FFill( void* self, double x ) {
     ((TH1F*)self)->Fill( x );
  }
}
```

- **To be resolved by targetting LLVM**
  - Low Level Virtual Machine
    - PyPy supported; can resolve C++ calls
    - http://llvm.org

Timing Results

- **Simple linear timing**
  - Code example from slide 6
  - Loop is part of the optimization
- **Preliminary Results:**
  - Python: **Real time 0:06:00, CP time 360.790**
  - PyPy: **Real time 0:00:14, CP time 14.550**
  - C++: **Real time 0:00:10, CP time 10.930**
  - C++ w/ stubs: **Real time 0:00:14, CP time 14.200**

=> *Very promising!*

Note: C++ w/ stubs (equal to the "Back-end Cheat") added for reference only

Conclusions

Success!

Shown proof-of-concept of enhancing a Python compiler with dictionary information, substantially improving the performance of HEP algorithmic code written in Python.

Future Plans

- **Target LLVM**
  - Remove need for C to C++ stubs
- **Package code outside PyPy**
  - Derive from and replace PyPy core classes
- **Automation, user-friendliness**
  - Installation (incl. PyPy and LLVM)
  - Auto-selection of optimization points
- **(Py)ROOT specific optimizations**
  - Unroll pythonizations (e.g. TTree, STL)

Resources

- **Code repository:**
  - http://nest.lbl.gov
- **Packages and documentation**
  - http://codespeak.net/pypy/
  - http://llvm.org/
  - http://www.scipy.org/
  - http://shed-skin.blogspot.com/
  - http://cern.ch/wlav/pyroot/

Bonus Material

TTree Ease-of-Use

- **Treat TTree as Python-style container**
  - Loop over TTree entries with *for*-loop ✔
  - Bounds and read-sanity checks ✔
  - Select slices (ranges)
- **Treat TTree as C-style struct**
  - Access branches as data members ✔
- **Optimize Python access and I/O**
  - Judicious use of caches
  - Read branches only on-demand

Optimization in PyROOT?

- **Main ingredients in inner loop codes:**
  - Numeric: math + builtins + arrays
    - Use NumPy with Psyco (JIT, x86 only)
  - ROOT extension library calls
    - Use dictionary info for static optimization
    - Dynamically change internal call settings
      - No autocast, return in local buffer, remove checks, etc.
- **Need fixed block of python code**
  - User defined, different optimization levels
  - But also such as THn::Fit and TFn::Draw

Prototyping Results (ROOT'07)

- **Effectiveness varies wildly ...**
  - Optimizations on builtins: no effect
  - Shortcircuit call stack: 45%
  - No check/cast on object ptrs: 10%
  - Drop bound method alloc: 25%
- **Effects are independent, so cumulative**
  - Total gain of about a factor of 3 in speed
- **Perhaps can be offered outside blocks**
  - Define configuration interface (or API)

Resources

- **Documentation**
  - Chapter 20 of the ROOT User's Guide
  - http://cern.ch/wlav/pyroot
- **Examples**
  - $ROOTSYS/tutorials/pyroot/*.py
- **Code Repository**
  - root.cern.ch/viewcvs/pyroot
- **Python optimizations**
  - http://wiki.python.org/moin/PythonSpeed/
  - http://www.scipy.org/