International Conference on Computing in High Energy and Nuclear Physics 2012 (CHEP2012) IOP Publishing
Journal of Physics: Conference Series **396** (2012) 052046 doi:10.1088/1742-6596/396/5/052046

Optimizing python-based ROOT I/O with PyPy's tracing just-in-time compiler

Wim TLP Lavrijsen
Computer Systems Engineer, Lawrence Berkeley National Lab.,
1 Cyclotron Road, BLDG 50B3238, Berkeley, CA, 94720-8147, United States

E-mail: WLavrijsen@lbl.gov

Abstract. The Python programming language allows objects and classes to respond dynamically to the execution environment. Most of this, however, is made possible through language hooks which by definition can not be optimized and thus tend to be slow. The PyPy implementation of Python includes a tracing just in time compiler (JIT), which allows similar dynamic responses but at the interpreter-, rather than the application-level. Therefore, it is possible to fully remove the hooks, leaving only the dynamic response, in the optimization stage for hot loops, if the types of interest are opened up to the JIT. A general opening up of types to the JIT, based on reflection information, has already been developed (cppyy). The work described in this paper takes it one step further by customizing access to ROOT I/O to the JIT, allowing for fully automatic optimizations.

1. Introduction

The two most widely used programming languages in High Energy Physics (HEP) are C++ and Python, with many millions of lines of validated code written in each. With commodity hardware changing towards multi- and many-core machines, these existing codes need to be adapted, and for Python, the most promising approach is to use the tracing just-in-time compiler (JIT, see section 2), from the PyPy project (see section 3). The Python language is particularly suited for adaptation, as it allows technology independent specification of scientific codes, leaving the hardware details to the JIT, in a manner that is not currently possible with existing C++ implementations. Furthermore, a strength of Python is that it is excellent in scheduling independent, serial, C++ codes in parallel. In other words, Python can be used to overcome platform dependencies that are inherent in legacy C++, while allowing continuing development in C++ for when that is desired.

Python has been called "executable pseudocode"[1] in that it is close to English-like notes of intent, such as may be developed on a piece of paper or whiteboard as a discussion-aid of what the code should do. This allows for sufficient technology-independent algorithmic codes. As an example, a listing comparing C++ and Python "analysis codes" is shown in figure 1. The code is of a typical loop over event-based data that is stored in the ROOT TTree format. The focus is not on the difference in length of the two codes, which is mainly caused by the extra boilerplate and that a code generator could take care of. Rather, the point is that all technology details are completely hidden, because the language offers the ability to have the run-time handle it. For example, types do not need to be declared, since they are not relevant to the algorithm. What this illustrates, is how Python's dynamic typing and extensible protocols allow the intent

of the code to be expressed cleanly, both to the human and machine readers. It is clear to the human reader that the Python code consists of an event loop over the input data, because that is literally what it says when read as English. It is also clear to the machine what should occur, as the proper hooks have been implemented into the run-time environment, and the right patterns are executed when called upon.

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

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

// read and use all data
Long64_t N = t1->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
```

Figure 1. Comparison of a C++ (left) and Python (right) "analysis" code, looping over data in a TTree structure as is commonly used in high energy physics.

Expressive as it is, there is one significant problem with this kind of Python code: the use of language hooks results in slow execution. This snippet runs 55x slower in Python than in C++, for example. Of course, this particular code was obviously chosen, by using direct access to data members and leaving out constructors in the I/O layer, precisely so that language overhead completely dominates. The difference is not nearly as stark in realistic programs, but even there it is of the order of 2-3x.

Can we do better? We know that C++-like performance can be gained from Python even when accessing C++ code across the language boundary with CPPYY (see section 4), so can the same techniques be applied to ROOT I/O? The answer is yes: the insight is that TTrees are much akin to class descriptions as used in CPPYY. A TTree describes a data structure that is potentially dispersed in memory, but with all the addresses known, it is no different than a description of an object layout and can thus be used as such.

This paper will first introduce the background of the technologies used, tracing just-in-time compilation, PyPy, and CPPYY; and then report on the performance achieved when applying these ideas and techniques directly to ROOT I/O.

2. Tracing just-in-time compilation

A tracing JIT differs from the better known "classical" JIT in that it does not optimize constructs that are known at the language level, such as methods that are found by a profiler to be executed often ("hot spots"). Instead, it observes actual executions, locates often executed code portions ("hot paths", not limited to single methods), collects linear traces of these executions, optimizes them, and compiles them to native code if applicable. Since a tracing JIT does not operate at

the language level, it can be applied to both binary as well as interpreted codes. An example of the former is DYNAMO [2], which operates on PA-RISC binaries: it is able to optimize, at run-time and in-flight, the native instruction stream that was already optimized during the original static compilation. It is able to improve *overall* execution speed, i.e. including the time spent running DYNAMO, it still beats total execution time over that of the statically optimized binary. DYNAMO can achieve these gains, because production binaries are build with a conservative choice of expected target platforms at the clients sites, because shared library boundaries and virtual function calls constrain static but not run-time optimizers, and because linear traces are much simpler to optimize than source code call graphs.

The basics of trace collection are outlined in figure 2. During execution, the tracer heuristically collects decision points, mostly by locating loops (the jump from 'F' back to 'A' in this example). After the point 'A' has been seen more than a set threshold number of times, the JIT collects the trace of execution (displayed by the green path), crossing function boundaries as need be. It then optimizes this path, and stores it for the next execution of 'A', leaving a guard (a new, internally used, decision point) on the condition that results in 'B'. If over time this guard gets hot as well, a new trace is collected, optimized, and inserted.

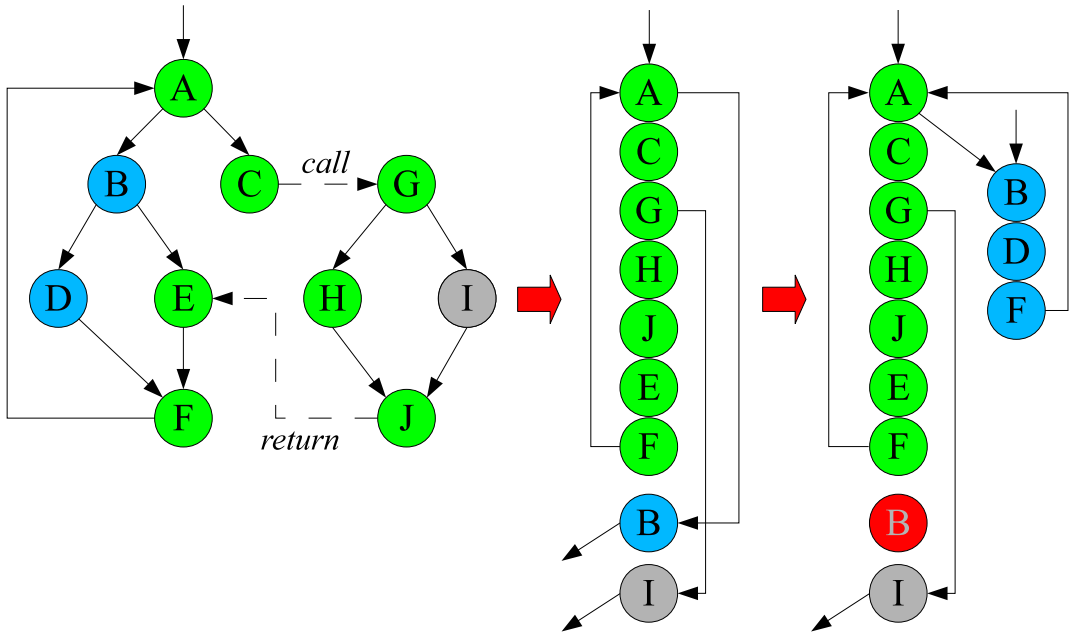

Figure 2. Outline of trace collection from a call graph: the green path is first selected, and after the guard on 'B' has turned hot as well, a second (blue) path selected.

A large range of benefits to HEP codes is immediately clear, given the uses, production platforms, dependence on data, and algorithmic nature of such codes:

- *Profiling on actual input data.* Rather than optimizing code on a "representative" data set, which they never are, especially not for a general purpose detector with many different event signatures, a tracing JIT optimizes the program conditioned on the actual input data to the current run. If different data leads to different execution paths, the traces, and hence the optimizations, naturally reflect that.
- *Compilation to actual machine features.* This is most important for data parallel instructions (such as SSE), the width, available register set, and labeling of which can

change from one CPU to the next. Static compilers could make use of these instruction sets, but the resultant binaries then require the target platform to support them. This is problematic in heterogeneous environments such as the grid, and hence binaries for grid-deployment are usually compiled to the oldest acceptable instruction set. By moving the compilation to run-time, this problem is avoided.

- *Inlining of function calls based on size and actual use.* There is a trade-off between call overhead and code bloat, which can result in memory bandwidth bottlenecks accesses, when inlining functions. The best way to decide which functions to inline, is by using profiling data. This can be done statically, but only if the runs and input data are representative. A tracing JIT makes the correct inlining decisions.
- *Instruction locality.* Modular frameworks that have many cross-module calls have their instruction sections scattered across main memory and have to rely on prefetching to prevent stalls due to cache misses. Prefetchers are very good with code that contains branches, but do not work as well with object oriented code and virtual functions in particular. Cache coherence is a given with compiled linear traces, however, and far fewer translation look-aside buffer entries are needed for the same code.
- *Trampolines removal.* The normal implementation on Linux for calls across shared library boundaries, is a procedure lookup table (PLT), that contains pointers to the functions that get resolved by the linker at run-time. Since cache lines span multiple entries in the PLT, and since the PLT is filled with trampolines and hence not a read-only code section, they become bottlenecks, especially if the CPU employs an exclusive cache strategy (as is the case in AMD CPUs). The trace cache is under control of the JIT and branch points are optimized for the relevant access patterns.
- *Putting heap objects on the stack.* To manage the lifetime of objects across different modules, they are allocated on the heap. Due to the modular nature of HEP codes, the number of allocations is so large, that the use of a different memory allocator often has a significant performance impact. Because traces are collected across function boundaries, lifetimes of many objects can be determined, and unnecessary construction/destruction pairs can be removed by placing the objects on the stack of the compiled trace.
- *Memoization.* Conversions of computation results can be judiciously cached, for example Cartesian versus Polar coordinates. In general, it is straightforward to add lazy evaluation to any function. Likewise, memory and CPU can be traded against each other on a per usage basis, rather than enforcing that choice at the level of classes for all possible use cases.

Other advantages of tracing JITs, but not necessarily relevant in HEP, are the smaller disk footprint compared to statically highly optimized (and inlined) code, and low-latency execution of code that is downloaded from a server.

3. The PyPy language development framework

Started in late 2002 with one of the main developers the author of PSCYO[3], a popular Python just-in-time compiler, PyPy has the following goals:

- A common translation and support framework for producing implementations of dynamic languages, emphasizing a clean separation between language specification and implementation aspects.
- A compliant, flexible and fast implementation of the Python language using said framework to enable new advanced features without having to encode low-level details into it.

To understand the PyPy project, it is important to keep its various levels of abstraction clear. PyPy is a framework to develop execution environments for dynamic languages in

Python. One such an environment written in the framework, is an implementation of Python that is compatible with the standard CPython interpreter. Other (preliminary) versions of interpreters implemented in PyPy exist for JavaScript, Prolog, Smalltalk, etc. From this description it should be clear that Python enters the discussion in two places: the language in which PyPy itself is written, and the implementation of a Python interpreter written in PyPy. It is important to keep a distinction between the two. In practice this works because PyPy can not translate the whole Python language, so its implementation language is actual a restricted subset of Python called "Restricted Python" or "RPython."

The implementation of an execution environment in PyPy is thus written in RPython, and it can run in the Python interpreter for rapid development. That, however, yields a slow implementation. Therefore, the RPython implementation is translated using a tool-chain into a target specific to the platform on which it will be run. In the translation process, such low-level details as memory management, the threading model, and object layout are automatically added. The result is then a platform-specific implementation of the interpreter, e.g. in compiled form, which can run at near-native speeds.[1] An important part of the translation, is the ability to generate a just-in-time compiler[4] for the interpreter being translated. For the JIT to work efficiently, the developer of the interpreter needs to provide a few hints, in particular the dispatch calls for certain language constructs such as loops, so that they can be easily detected by the JIT. With the JIT-generation enabled, the final result is a platform-specific implementation of the interpreter, with a built-in JIT.

4. The cppyy project

It is expected that it will always be important to handle cross-language calls between C++ and Python, if only to use legacy C++ codes. The PyROOT[5] project is the de facto standard in HEP for binding a C++ library to be used from Python, but it relies on the standard Python C-API for this. In this API, there is the concept of a "PyObject", which has a reference count, a pointer to a type object, and some payload. The API allows extraction of the low-level information from the payload for use in the C++ call, and can repackage any results from the call. This marshalling is where the bulk of the time is spent when dispatching between the two languages. To be absolutely precise, most C++ extension module generators produce slow dispatches because they don't handle overloads efficiently, but even in there, they still spend most of their time in the marshalling code, albeit in calls that fail before trying the next overload. For this reason, PyROOT hashes the types of the call arguments to be able to make a quick selection of the correct overload, when it has succeeded once.

In PyPy, speed is gained by having the JIT unbox objects into the payload only, allowing it to become part of the compiled traces. If the same marshalling APIs were used, the JIT would be forced to rebox the payload, hand it over through the API, only to have it unboxed again by the binding. Such an approach is extremely inefficient. The goal of the cppyy project, is to provide the bindings while keeping all code transparent to the JIT until the absolute last possible moment, i.e. the call into C++ itself. This allows the JIT to (more or less) directly pass to C++ the payload it already has, with an absolute minimal amount of extra work. In the extreme case when the binding is not to a call, but to a data member of an object (or to a global variable), the memory address is delivered to the JIT and this results in direct access with no overhead. Note the interplay: cppyy in PyPy does not work like a binding in the CPython sense that is a back-and-forth between the interpreter and the extension. Instead, it does its work by being transparent to the JIT, allowing the JIT to dissolve the binding.

[1] To be absolutely clear: it is the translated interpreter that runs at near-native speeds, i.e. at CPython speeds, meaning that user code executed on it runs at speeds of the normal Python implementation. Further performance gains are only made possible by the JIT.

It is important to note that there is no performance penalty in creating extension classes dynamically rather then generating code for extension modules and compiling those: Python classes and functions are always created dynamically by the interpreter, even for "compiled" extension modules. Furthermore, by creating extension classes dynamically, CPPYY can add so-called "pythonizations," or the automatic integration of known C++ classes or language features into a Python protocol, on the fly, purely based on interface seen, such as for example a C++ STL(-like) container providing iterators or indexing.

5. Benchmark results

To ensure that the exact same "analysis" code can be run on PyPy as on CPython, a PyROOT compatible module, called CppyyROOT.py, was developed. With that available, an implementation for describing TTrees, which makes use of the same techniques that allow the JIT access to class data members through CPPYY, was added at the interpreter level (i.e. before the translation, making it transparent to the JIT). The TTree instances that the user can interact with on the Python prompt, acquire additional variables, which represent the data objects read, but are otherwise unchanged and are still usable wherever TTree instances are expected in either C++ or Python.

The result is a more than 20x speedup over the original CPython execution. This still leaves a difference of 2.7x compared to C++, but as as explained in section 1, the code snippet was chosen to maximize the language overhead. In realistic programs, if the code is I/O bound, if CPU-time is spent in mathematical codes as well, if data classes require constructors, or if they are transient/persistent separated, etc., etc., that difference of 2.7x amortizes quite well.

Nevertheless, it is still good to find out why a difference persist: the reason is that the JIT is conservatively leaving guards in place, in case the data members handed to the TTree get removed, or change type. The overhead of the guards, which of itself is very small, causes such a large difference, because the overhead for access to the data members in C++ is virtually zero. It is indeed possible that the user removes or alters the exposed data members, so the JIT is correct in its conservative approach. In a follow-up, the dictionary set of the TTree instances could be made read-only, effectively "freezing" them, allowing the JIT to optimize the guards out of existence. This would impose an inconvenience to the user who does want to remove this internal data, so this approach should be configurable, if there is indeed a relevant use case to be made where removal is necessary.

Further improvements are now possible. In particular, since the trace sees which elements are read, it is possible to only activate those branches on the TTree instance that are used in the trace, saving on time spent in reading bytes from disk. It will now also be possible to perform other code transformations, for example turning this serial code into code that runs PROOF[6], allowing automatic parallelization on the event level.

6. Conclusions and outlook

The Python language is one of the two most popular languages in HEP. It allows a scientific description of intent, that is free from technology details. The expressiveness of the language comes with a significant cost, however. Utilizing the PyPy tracing JIT makes it possible to remove this cost by compiling the high level description down to machine code to run at native speeds. ROOT I/O is setup to be optimally used from C++ and poses significant problems for performant use from Python: a worst case scenario has been shown to be 55x slower than the equivalent C++ code. Taking the ideas and techniques from CPPYY, which provides C++ bindings for the PyPy project, and applying them to ROOT I/O from Python, this paper has shown that it is possible to achieve a performance increase of 20x over CPython. A slowdown of 2.7x compared to C++ still remains and there are some ideas to remove this final overhead, by "freezing" the TTree instances, but in realistic programs that may not even be necessary.

With this good performance in place, the next steps are to enable usage-based selection of data read, and to apply transformations that turn the serial code as written into parallel code, for example by making use of PROOF for event-level parallelization.

References

[1] Lutz M 2006 *Programming Python* (O'Reilly Media)
[2] Bala V, Duesterwald E and Banerjia S 1999 *Transparent Dynamic Optimization: The Design and Implementation of Dynamo* (HP Laboratories Cambridge, HPL-1999-78)
[3] Rigo A 2004, *Proc. of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (Verona)*, p 15-26
[4] Bolz C F, Cuni A, Fijalkowski M and Rigo A 2009, *Proc. of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (Genova)* p 18-25
[5] Generowicz J, Lavrijsen W T L P and Marino M 2004, Reflection-Based Python-C++ Bindings, *Proc. Intl. Conf. on Computing in High Energy Physics (Interlaken)*
[6] `http://root.cern.ch/drupal/content/proof`