PROCEEDINGS OF SCIENCE

The Role of Interpreters in High Performance Computing

Axel Naumann*CERN

E-mail: axel.naumann@cern.ch

Philippe Canal
Fermilab
E-mail: pcanal@fnal.gov

Compiled code is fast, interpreted code is slow. There is not much we can do about it, and it's the reason why interpreters' use in high performance computing is usually restricted to job submission. We show where interpreters make sense even in the context of analysis code, and what aspects have to be taken into account to make this combination a success.

XII Advanced Computing and Analysis Techniques in Physics Research
November 3-7, 2008
Erice, Italy

*Speaker.

PoS(ACAT08)065

1. The Context

High performance computing favors optimized algorithms that are then processed by a computer as quickly as possible. C++ has been a great success for that, enabling large collaborations to collect a large code base without too much overhead. Even more important is the fact that C++ *allows* code to be highly optimized: not all code needs to run with as few CPU instructions as possible, but where performance matters C++ can be brought up to FORTRAN's or C's performance. Highly efficient (presumably expert) code and less efficient code can thus coexist.

Interpreters have always been used to steer the high performance processes: To submit them to batch systems, to determine parameters, or to prepare input files. There has thus been a clear separation of the high performance analysis and the low performance preparation. Such an interpreter used to be called a *shell*; typical examples being bash or tcsh. With the advent of Python as a "glue" language, much of the old shell code has been migrated to interpreted languages.

Interpreted algorithms are slower than compiled ones, because of the extra runtime cost incurred by generating most CPU instructions at runtime.[1] Nevertheless it is faster to write and test algorithms in interpreted languages. This is probably a major reason of the success of interpreted languages also in the context of high performance computing: many of the LHC experiments' algorithms are both designed and used in interpreters.

2. Algorithmic Configuration and Interpreters

Configurations are non-compiled input to algorithms, e.g. statements in a text file that is read by a program. Many formats exist; simple key / value maps like show in listing 1, or XML like `<jetAlgo name="Cone08"/>`. They facilitate the change of input parameters by allowing to edit a text file instead of rebuilding the whole program.

These configurations are often going beyond defining simple numerical values anymore; instead parts of the algorithms themselves are defined by configuration settings. This simply acknowledges the physicists' reality: algorithms often need to be changed several times until the optimal one has been found, and algorithms are used in several derivations (e.g. signal versus background). But rebuilding software embedded in experiments' frameworks can take as much as half an hour for a single line of changed source code, simply due to an overwhelming amount of dependencies. Instead, physicists prefer to change a text file and leave the binary untouched.

An actual example of an algorithmic configuration can be seen in listing 2. Here, a simple syntax needs to be extended to allow for more complexity, e.g. to connect subroutines and to define data flow. This syntax can in fact be seen as its own interpreted language; after all, it defines algorithmic behavior, just like interpreted languages do. Extending a simple configuration syntax to a more complex one, i.e. creating a custom interpreted language, requires also a parser (or interpreter). It needs to be written, tested and maintained; it needs to handle errors, allow

```
minJetEt: 12
jetAlgo: Cone08
```

Figure 1: Example for a simple key / value map based configuration.

```
+postZeroJets.Run: NJetsCut(postzerojets) \
                   VJetsPlots(postZeroJetPlots)
postzerojets.JetBranch: %{VJets.GoodJet_Branch}
```

Figure 2: Example for an algorithmic configuration.

for debugging — just like any real interpreted language. It forces users to learn a language with an extremely specialized area of application; a language that will be seen as useless to learn by many users. The custom language will also force users to learn and deal with new tools e.g. for debugging, if these tools exist at all.

We believe that this should instead be done by one of the existing interpreted languages. This allows users to learn a more useful language, it allows the reuse of existing parsers and interpreters, and basically all existing interpreter languages can cover any possible scenario even for algorithmic configurations. Some interpreters offer libraries that allow access to compiled code, e.g. to call compiled C++ functions. This in turn allows extremely powerful algorithmic configurations, where the configuration can immediately interact with compiled code. Interpreted languages thus boost the user's agility by being more expressive, and by reducing the threshold for recompilation of binaries.

3. Interpreters in Software Frameworks

Usually, several source files of an experiment's software framework need to be changed to implement a new reconstruction algorithm, or to set up a physics analysis. This is done by regular physicists; it thus needs to be as easy and fail safe as possible. Interpreters can help considerably with this. They allow to localize changes: instead of changes in files scattered across the framework's many directories only a small source file (*script*) needs to be maintained. It gives a modular test bed that increases the developer's confidence of control of the software, simply because the code changes are clearly arranged and have less side effects. Changes are also easier to track, e.g. through code versioning systems that can be checked into by regular physicists: after all, they do *not* overwrite the experiment's code but implement their own scripts instead. Probably most importantly, these modular pieces of code are much easier to communicate, which in turn helps in the validation of algorithms.

4. Interpreters and Interpreted Languages

Traditionally, interpreters came with special interpreted languages that have a simplified syntax. They were easier to learn and allowed simplified data handling: C++ code as shown in listing 3 becomes a lot shorter if written in interpreted languages, as shown in listing 4.

The lack of syntactical precision is often paid for by reduced runtime performance. The code is interpreted on demand which allows it to use constructs defined by context: e.g. variables can be declared without a type; their type is determined by whatever is assigned to them. Compiled languages, on the other hand, usually require variables to be defined already at compile time. While

```
vector<Electron>* ve = 0;
tree->SetBranchAddress("Electrons", &ve);
tree->GetBranch("Electrons")->GetEntry(0);
for (int i=0; i<ve->size(); ++i) {
   Electron* electron = ve[i];
...
```

Figure 3: Example loop in C++.

```
foreach electron in tree.Electrons:
   ...
```

Figure 4: Example loop in a hypothetical interpreted language.

this makes code easier to write and to reuse, the validity of code can often only be checked at runtime, when a variable's type is defined.

The use of specialized interpreted languages also means that a new language has to be learned. Their loose syntactic precision and the lack of focus on performance is often seen as a language's advantage: once users of interpreters have written enough code in an interpreted language it takes a conscious effort to move to compiled languages — it is just too easy to write code in a syntactically imprecise (i.e. runtime-defined) way. Specialized interpreted languages thus cause a problem that can already be seen in some LHC code: there is a push towards interpreting algorithms[2], ignoring the CPU time cost this incurs.

5. Interaction Of Interpreters With Compiled Code

As we have shown, the interaction of interpreted code with compiled code is a fundamental property of interpreters in high performance computing. Unfortunately it is also one of the most neglected aspects. The interaction can be seen as a communication process: interpreters must be able to make use of compiled code, and compiled code must be able to interpret code by means of the interpreter's API. Another aspect of the interpreter-compiled code interaction is reflection which we will cover in section 6.1

5.1 From Interpreted To Compiled Code

Interpreters need to be able to call compiled code to be a useful tool for rapid prototyping in a complex code environment, e.g. within the LHC experiments' software frameworks. There are several attempts to ease the calling of compiled code from interpreters: python offers PyROOT for the C++ data analysis framework ROOT[3] and the LHC experiments' frameworks (a project exists to give access to ROOT from any of Microsoft's Common Language Runtime (CLR) languages[4], e.g. IronPython) bindings to ROOT's classes exist for Ruby, Perl and others can be used with the help of SWIG[5].

This can justify the use of languages different from the software framework's language in restricted amounts. What we see, though, is that it is a one way street with a dead end: it is much

```
gROOT->ProcessLine("Klass::Gimme()");
```

Figure 5: Invoking CINT from compiled code.

```
// loads libKlass.so / libKlass.dll /...:
root [0] gSystem->Load("libKlass")
root [1] Klass* k = Klass::Gimme()
root [2] k->Say()
```

Figure 6: CINT and user code: `Klass::Gimme()` becomes a function call in `Klass.so`.

more difficult to migrate code written in any of these languages into compiled code, simply because their languages do not offer compilers.[1] Where they do (like for C#) the resulting binaries are incompatible with the experiment's software framework. We believe that for software frameworks based on C++ this can only be solved by being able to interpret C++.

5.2 From Compiled Code To Interpreters

It has proved to be a key feature for interpreters to be called from compiled code. ROOT implements the signal slot mechanism, components of the I/O subsystem that are not time critical, and most importantly its plugin-system based on the C++ API of its C++ interpreter CINT[7]. CINT allows to interpret C++ from compiled code as shown in listing 5 and to call any functions in instrumented user libraries as shown in listing 6. This instrumentation is combined with the one needed to retrieve reflection information as presented in section 6.1.

5.3 Migration of Interpreted To Compiled Code

Another aspect of the transition from interpreted to compiled code is the migration of interpreted code into libraries. This can be useful for performance critical algorithms, but also to use the compiler to check the code's correctness with "standard" compiler error messages. The transition to compiled code can be simplified when using an interpreter, as most of the information needed by build systems is already available: include paths to find header files, library dependencies, the name of the compiler and linker can all be accessed from an enhanced interpreter.

ROOT and CINT implement this feature in an online compiler called ACLiC. While `.L myCode.C` loads a source file into CINT for interpretation, `.L myCode.C+` compiles and links that source file as a shared library before loading it into CINT. This replaces platform dependent build systems including dependency tracking; it also enhances the user code to provide reflection data and to make the user code accessible from the interpreter.

6. C++ Interpreters

The idea of CINT was to provide a minimum amount of C++ necessary to interact with a

[1] What is commonly referred to as Python's compiler is in fact a bytecode compiler; its output still needs to be interpreted to be run. A dedicated Python compiler exists[6] but it introduces many restrictions to Python and is to our knowledge not widely used in High Energy Physics.

C++ system. Offering a C++ interpreter has proved to be so successful that this basic set of C++ was soon seen as insufficient. We are addressing this issue with a new interpreter project, see section 6.3. C++ interpreters are very rare; we know of only two actively developed interpreters. As we have already pointed out, the major advantage of C++ interpreters compared to other languages (in a C++ environment) is the seamless integration in the compiled software context. If their parsers are powerful enough they can also be used to gather reflection data which can then be used for interpreters and serialization.

Most importantly, C++ is a prerequisite for most experiments' software frameworks anyway; it does not add the requirement of learning a new language, and all knowledge gained with the interpreter can be applied unchanged to the framework. The interpreted code can be easily migrated to the software frameworks once development is finished; it integrates seamlessly with the framework and e.g. ROOT. And still: thanks to the interpreter, during C++ development the turn-around of editing and running code is as quick as with any other interpreter.

6.1 Interpreters And Reflection

Information about available types and members (data as well as function) is called reflection data. It is paramount for serialization of C++ objects and for interpreter access to compiled code. Another contribution discusses many of its details[8]; here we only introduce the properties most relevant for interpreters.

A key ingredient of an interpreter is its parser. It defines its ability to understand a language, both for interpreted code and to read header files. The latter is needed to extract the available types and members so they can be used e.g. from an interpreter. CINT can do just that; its limitations with understanding C++ provoked a second path for gathering reflection data to be used with CINT. This alternative approach uses *GCCXML*[9] (based on GCC's parser) to extract reflection data and a python script (*genreflex*) to convert it to a format that is compatible with CINT. It adds a considerable compile time overhead due to the creation of a temporary XML file which is then parsed by a python script.

6.2 Enabling Library Calls From Interpreters

To enable function calls into libraries, the interpreter needs to know about the available functions and their parameter types. It also need to be able to do a C++ call (i.e. respecting the operating system's ABI) into the library. This is highly platform dependent; the most transparent way to to generate a function stub which is called to forward the call. This stub is typically a C function taking a vector of `void*` as parameters; the parameters are then converted to the types expected by the actual function to be called.

For both CINT and GCCXML / genreflex, reflection data and the function stubs are compiled and linked C++ code, either integrated into the original library to build an instrumented library, or as a separate library to be loaded when the interpreter needs to access the library. Upon loading the reflection code initializes memory data structures from data stored in the C++ file. This means a duplication of strings in memory: they are part of the library and are again copied into heap for the memory data structures. Both CINT alone and GCCXML / genreflex increase the size of a library because by approximately a factor 2, they also come with a non negligible memory usage.

There are ways to reduce the cost of the library instrumentation to generate and store the reflection data needed for interpreters. All of them applied, the cost in library size can be reduced to about 10% of the original library's size; the build time overhead is approximately comparable to compiling two source files.

6.3 Cling: An LLVM-Based Interpreter

Instead of optimizing existing concepts we decided to develop a new C++ interpreter called *cling*. It will provide a remedy to the issues of increased memory usage, library size increase, the inflation of compile time, the lack of support for C++, and that will improve the link between compiled and interpreted code. The fundamentally new feature compared to CINT is that it is based on a compiler infrastructure instead of providing all of an interpreter's ingredients itself.

The compiler infrastructure we have chosen is *LLVM*[10] with its C++ front-end *clang*. They feature a C++ API, very modular design, an already existing bytecode interpreter; they are designed to work on a wide range of platforms, and most importantly they are open source projects with production quality. It is a strong competitor to GCC both in performance of the compilation and of the generated binaries. They are hosted at the University of Chicago, supported by Apple, Adobe, and many others.

One of the most important features of LLVM is its just-in-time (JIT) compiler: on selected platforms (Linux, MacOS and Windows being the most relevant ones) code can be compiled in memory, incorporating all optimization steps available through LLVM. It will allow us to replace stubs for function calls by JIT-based in-memory-linking, which in turn reduces the size of the instrumentation libraries.

The clang C++ front-end will allow us to access parsed code, to analyze it, to append to it, and even to modify it. We will use the precompiled header feature of clang/LLVM to store reflection data. This speeds up the instrumentation of libraries considerably as there is no need to compile or link, let alone generate and parse XML with python. The memory foot print will also be reduced because the precompiled headers are loaded as heap data structures which can be freed once the reflection data has been extracted.

7. Conclusion

We believe interpreters play an important role in high performance computing, despite the fact that they waste CPU cycles for data processing compared to compiled code. Especially in the context of gluing applications together, setting – even algorithmic – configuration parameters, and code development, interpreters can be an ideal ingredient.

While it is intriguingly easy to write code with dedicated interpreter languages, we have shown that these languages have a very limited area of application. They are often used outside these limited areas, costing experiments a large fraction of CPU time. We believe that the interpreter must match the compiled software framework to be a viable addition, to allow performance critical code to be seamlessly migrated to the framework. Only then will the experiments be able to analyze their data within the planned amount of CPU power, still offering agile development enabled by interpreters.

References

[1] Debian, *The Computer Language Benchmarks Game*, http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=all

[2] Atlas Collaboration, *Atlas Reconstruction Software*, http://alxr.usatlas.bnl.gov/lxr/source/atlas/Reconstruction

[3] R. Brun and F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, in *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996*, Nucl. Inst. & Meth. in Phys. Res. A **389** (1997) 81-86. See also http://root.cern.ch

[4] G. Watts, *ROOT.NET: Making ROOT accessible from CLR based languages*, CHEP 2009. See also http://rootdotnet.sourceforge.net

[5] D. M. Beazley, *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*, 4th Annual Tcl/Tk Workshop, Monterey, CA. July 6-10, 1996. See also http://www.swig.org

[6] *PyPy - a Python implementation written in Python*, http://codespeak.net/pypy/dist/pypy/doc/translation.html

[7] M. Goto, *C++ Interpreter - CINT*, CQ publishing

[8] A. Naumann and P. Canal, *C++ and Data*, in proceedings of *ACAT 2008, Sicily*, PoS(ACAT08)073

[9] B. King, Kitware, *GCC-XML, the XML output extension to GCC*, http://gccxml.org

[10] Ch. Lattner and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*