Implement a GlobalModuleIndex in ROOT and Cling

Arpitha Raghunandan
April 8, 2019

1. Abstract

Most software requires some libraries. For each library, one needs to access both its interface (API) and its implementation. In the C family of languages, the interface to a library is accessed by including the appropriate header files. The implementation is handled separately by linking against the relevant library. Modules provide an alternative, simpler way to use software libraries that provide better compile-time scalability and eliminates many of the problems involved in using the C preprocessor to access the API of a library.

ROOT is at the heart of the research on high-energy physics (HEP). Originating at CERN, it is a modular scientific software toolkit or a framework for data processing. HEP physicists use ROOT applications to analyze their data or to perform simulations. ROOT can be used to store, access and mine data. It displays results graphically and can be run interactively with the help of Cling C++ interpreter. Industry standard C++ modules API is implemented in Clang C++ compiler, and Cling is a C++ interpreter originating at CERN. ROOT calls the Cling API to load C++ modules.

ROOT has several features which interact with libraries and require implicit header inclusion. These headers are often immutable, and reparsing is redundant. C++ Modules are designed to minimize the reparsing of the same header content by providing an efficient on-disk representation of C++ code. Although C++ modules support in ROOT has been implemented in the last few years, there is still room for performance improvement, and GlobalModuleIndex implementation is one such possible solution. It is a mechanism to create the table of symbols and PCM names so that ROOT will be able to load a corresponding library when a symbol lookup failed.

The aim of this proposal is:

- To implement a GlobalModuleIndex, which is expected to improve ROOT's performance by speeding up its startup time.

2. Introduction and Goals

This project needs to be implemented as part of ROOT and Cling, as ROOT is loading C++ Modules by calling Cling API. It has been partially implemented, and the project aims at reducing the number of modules that need to be loaded.

Currently running `root.exe -l -q tutorials/hsimple.C` with `runtime_cxxmodules=On`, will load 80+ modules. One of the mentors of this project, Vassil Vassilev has provided a raw implementation of the same which reduces the number of modules loaded to around 50. This project aims at reducing the number of modules loaded to be less than 10.

The current GlobalModuleIndex implementation is deficient because the Global Module Index is essentially an on-disk hash table which keeps the mapping `identifier->module`. Consider,

```
module A {
  header "A.h"
  export *
}

module B {
  header "B.h"
  export *
}
```

```
// A.h
class Foo;
```

```
// B.h
class Foo {};
```

The Global Module Index will contain `Foo->A, B`. Currently this implementation loads both module A and module B. Usually, every time we look up an entity, we require its definition. This means that we should load module B and not module A in the above example. This project aims at writing the identifier mappings in the Global Module Index in such a way that the definitions are stored first, which will help in reducing the number of modules to be loaded.

The tasks to achieve the above goal are:

1. Understand the lazy loading of PCMs in GlobalModuleIndex in the current implementation.
2. Implement storing of definitions first in the GlobalModuleIndex.
3. Enhance the implementation of a callback in Cling, which gets a callback when an identifier lookup fails.
4. Deprecate startup modules loading step in ROOT.
5. Test the new implementation and provide performance measurements.

3. Implementation plan

This section details the procedure to be followed to implement the above tasks.

3.1. Enhancing prototype implementation of GlobalModuleIndex

Study the current design and implementation of GlobalModuleIndex and find out why PCMs are being loaded excessively. Design and implement GlobalModuleIndex such that the definitions are stored first. Enhance the design and implementation of the callback in Cling, which gets a callback when an identifier lookup fails. Test the new implementations and fix any bugs that may arise.

3.2. Deprecating startup modules loading step in ROOT

Understand the startup modules loading step in ROOT. Deprecate the same and provide an implementation which helps load the minimum required modules at the startup time.

3.3. Testing the new implementation and providing performance measurements

Complete all implementations and develop tests for performance measurements. Compare the memory footprints of the new and old implementations.

4. Timeline

Week:

1. **May 7 – May 26 (Community Bonding Period)**
   Understand the codebase. Understand what must be done over the next few months. Study the current design and implementation of GlobalModuleIndex. Communicate with mentors and other members of the community.
2. **May 27 – June 2**
   Investigate the GlobalModuleIndex prototype implementation and find out why PCMs are being loaded excessively. Come up with the design of new procedures for storing the definitions first in the GlobalModuleIndex.
3. **June 3 – June 9**
   Implement the new procedures for storing the definitions first in the GlobalModuleIndex.
4. **June 10 – June 16**
   Continue implementation of the new procedures for storing the definitions first in the GlobalModuleIndex.
5. **June 17 – June 23**
   Test the new procedures for storing the definitions first in the GlobalModuleIndex. Provide initial documentation for the work done.
6. **June 24– June 30**
   Enhance the design of callback in Cling, which gets a callback when an identifier lookup fails.
7. **July 1 – July 7**
   Enhance the implementation and test of the callback in Cling, which gets a callback when an identifier lookup fails.
8. **July 8 – July 14**
   Design the deprecating startup modules loading step in ROOT.

9. **July 15 – July 21**
   Implement the deprecating startup modules loading step in ROOT.
10. **July 22 – July 28**
    Test the implementation of the deprecating startup modules loading step in ROOT.
11. **July 29 – August 4**
    Test the overall program and make sure the implementation works correctly and fix bugs which emerge during testing.
12. **August 5 – August 11**
    Finalize the implementation and land patches to upstream.
13. **August 12 – August 18**
    Evaluate the performance of the new implementation.
14. **August 19 – August 25**
    Write detailed documentation about this program.
15. **August 26 – September 1**
    Create a report for the final review.

5. Personal Information

- Name: Arpitha Raghunandan
- Residence: India
- Email: 98.arpi@gmail.com
- Affiliation: IT student at National Institute of Technology Karnataka, Surathkal, India
- Timezone: UTC+5:30

5.1 Availability

I will be fully available during the GSoC period and able to spend over 40 hours per week because I have no other commitments at this time. I'm willing to spend a significant amount of time on this project.

5.2 Previous experiences in ROOT

Related to this project, I have made the following pull requests to the ROOT repository as per the suggestions of potential mentors, Yuka Takahashi, Oksana Shadura and Vassil Vassilev:

1) Added additional headers to stl.modulemap: https://github.com/root-project/root/pull/3531

2) Reducing ExcludeModules one by one:
https://github.com/root-project/root/pull/3580
https://github.com/root-project/root/pull/3635

I believe that my experience with the codebase while working on the above PRs will help me in completing this project.

5.3 Previous experiences and Skills

I have experience in object-oriented programming, with languages C++, Python and Java. I have also worked with procedural programming in C. I have taken up several courses in the field of computer science at my University, namely:

- Data Structures and Algorithms, Design and Analysis of Algorithms
- Graph Theory, Linear Algebra and Matrices
- Paradigms of Programming
- Unix and Operating Systems
- Networks and Communication
- Number Theory and Cryptography

5.3 Motivation

I was introduced to the field of Computer Science through Cryptography in a three week camp as a part of the Duke University Talent Identification Program Summer Studies Program. This sparked my interest in related areas of computer science. Last summer, I learned about reverse engineering during a program conducted by the IEEE Student Branch at my Institute. I learned assembly language through this. I was also introduced to CTFs (Capture The Flag, security contests), and I enjoyed participating in them.

I took up a group project to implement the Return Oriented Programming exploit last semester. Through this project, I learned about various security vulnerabilities and measures to prevent them by reading many blogs and papers. I learned about executables and how to disassemble them using open source frameworks like Capstone. This further introduced me to the basics of the compiler technology. As a part of a course in my Institute, I learned to build a basic interpreter using yacc and lex.

I wanted to contribute to Open Source on something related to one of the above fields. When I came across this project, I realized it was a perfect fit for me. The

basics of compiler technology that I have learned has made me interested in this field and motivated me to get a deeper understanding of the same. I hope to use whatever I have learned so far while contributing to this project. Further, I would learn a lot more about C++ compiler and interpreter technology along the way. It would be great to contribute to large Open Source projects like ROOT and Clang/LLVM.