Google Summer of Code

Optimize ROOT use of modules for large codebases

Final Report

Jun Zhang (jun@junz.org)

Overview

For some very large C++ projects, the compilation time becomes a huge pain. One major factor of this issue is redundant header parsing. (Imaging every time you do `#include "foo.h"`, it just copies and pastes everything and starts reparsing.) This is more problematic for systems such as ROOT because the code is parsed at runtime. To solve this issue, we have adopted Clang modules in ROOT, which alleviates the problem to some extent. However, the current module integration also has its limitations. For example, one source of performance loss is the need for symbol lookups across a very large set of modules. Take the example below:

```
root [0] edm::X;
```

When ROOT sees `edm`, it will immediately pull up all modules containing it, which is totally unnecessary since we only care about `X`. So my project this year is aiming to eliminate the unnecessary lookup for identifiers like a namespace to reduce memory pressure.

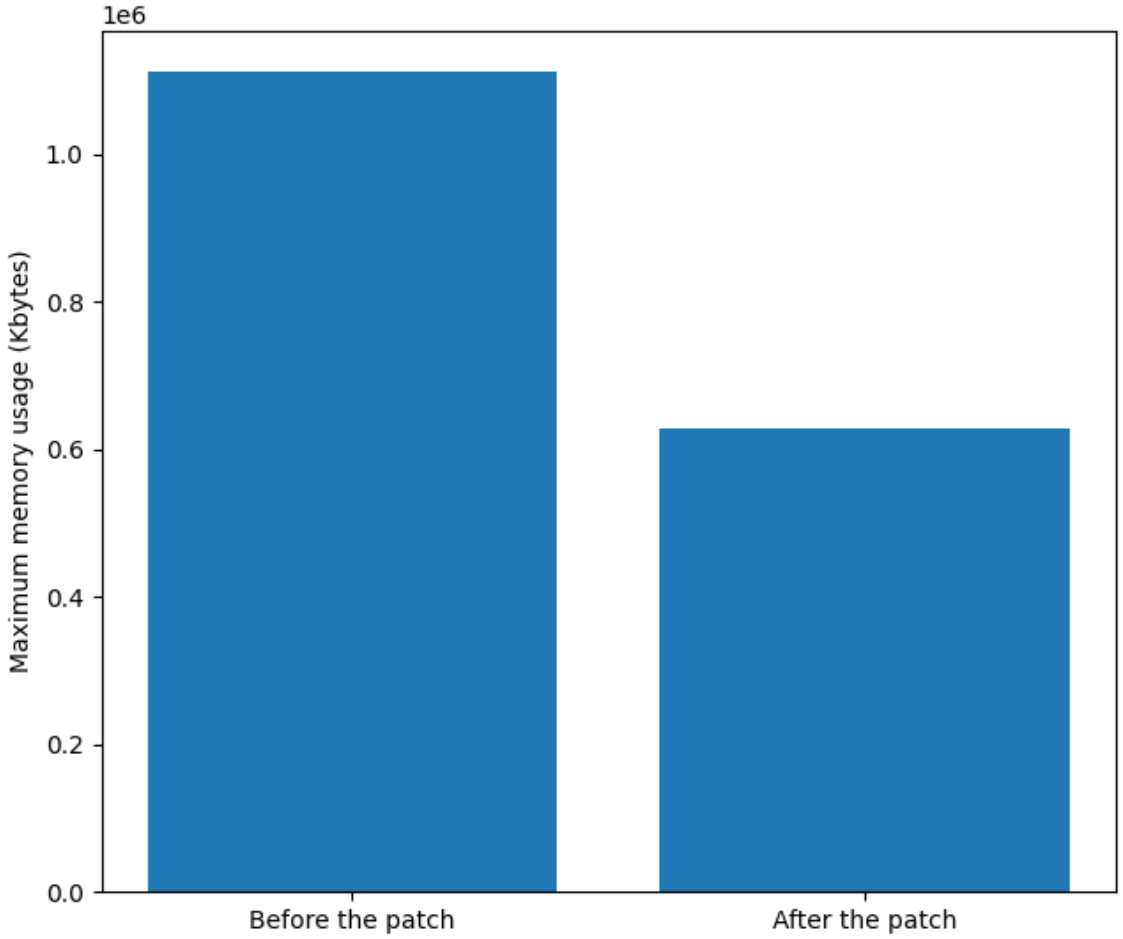
The Work I have done

At the very beginning of the project, I tried to eliminate all namespaces identifiers lookups as it's the most straightforward idea. But we soon ran into some troubles. The problem is that we lost the ability to recognize identifiers inside those nested namespaces, which is a very common case:

```
error: no member named 'MakeTrivialDataFrame' in namespace 'ROOT::RDF'
   auto d_s = ROOT::RDF::MakeTrivialDataFrame(nEvents);
              ~~~~~~~~~~~~^
```

This failure looks hard to solve so I was thinking: Why not achieve our goals step by step and only take top-level namespaces into consideration? So I did. The test results look good but when it comes to the performance improvement, Ugh... the results don't look very satisfying. The reason is simple, as there're not too many top-level namespaces, so we can't get a significant memory decrease.

We seem to touch some dark corners in ROOT and are trapped by the unfathomable semantics of C++. During that period, I explored many directions but none seems to work. My mentors and I had some deep discussions, and finally, we came to a new approach: If we cannot ignore those namespaces, why not create a new module that contains all namespaces forward declarations? Because the manually generated module just consists of forward declarations, it's cheap to preload, and ROOT can have all essential knowledge about namespaces with it. So here's the PR. The performance improvement also looks very promising:



```
Singularity> ROOTDEBUG=7 root.exe -l -b -q $ROOTSYS/tutorials/hsimple.C |&
grep "Loading" | wc -l # The mainline ROOT with CMSSW integration
180
Singularity> ROOTDEBUG=7 root.exe -l -b -q $ROOTSYS/tutorials/hsimple.C |&
grep "Loading" | wc -l # After applying our patch
52
```

In short words, we successfully reduced the amount of memory in `CMSSW` by half (from 1.1GB to 600MB) for simple workflows like `hsimple.C`, and the modules we're loading for it reduced from 180 to 52.

I also made some presentations about my work to the team, and they can be found in:

- Road Map of Optimize ROOT Use of Modules For Large Codebases
- Current Working Status of Optimize Modules Usage For Root

Post GSoC

However, it's not enough. I have done some deep profiling about Clang, and found out the bottleneck:

```
% total        Total        Calls  Function
   99.3  444'432'413       74'169  main [4]
   92.3  412'998'941       29'027  llvm::SmallVectorBase::grow_pod(void*,
unsigned long, unsigned long) [5]
   89.2  399'211'718        7'870
clang::CompilerInstance::loadModule(clang::SourceLocation,
llvm::ArrayRef<std::pair<clang::IdentifierInfo*, clang::SourceLocation> >,
clang::Module::NameVisibilityKind, bool) [6]
   89.2  399'058'838        7'644  clang::ASTReader::ReadAST(llvm::StringRef,
clang::serialization::ModuleKind, clang::SourceLocation, unsigned int,
llvm::SmallVectorImpl<clang::ASTReader::ImportedSubmodule>*) [7]
   88.8  397'250'710        8'023
cling::Interpreter::loadModule(clang::Module*, bool) [8]
   88.7  397'152'678        7'901
clang::Sema::ActOnModuleImport(clang::SourceLocation, clang::SourceLocation,
clang::SourceLocation, llvm::ArrayRef<std::pair<clang::IdentifierInfo*,
clang::SourceLocation> >) [9]
   84.0  376'053'039       12'354  TRint::TRint(char const*, int*, char**,
void*, int, bool, bool) [10]
```

From the above, we can see the hottest method is `SmallVectorBase::grow_pod`. That is actually caused by a problematic model: Clang always preallocates a huge buffer that is enough to contain all `SourceLocation` in a module, even without use.

The easiest solution in our mind is: Can we just get rid of `SmallVector` and use some more suitable data structures? So we tried to `SparseVector` which behaves like a normal vector but saves more spaces when the elements are sparse. Unfortunately from the performance results we see almost no improvement and we didn't have enough time to figure out why.

In conclusion, we pinpointed the real issue in Clang. Rather than trying to mitigate the problem on the side of ROOT, I've started to talk with Clang modules code owners and plan to continue the work and cooperate with the upstream to further improve modules in ROOT. Hopefully, we can solve this longstanding issue and make Clang better.

Other work

Except the modules work, I also done a significant contribution to `clang-repl`, which is an upstream version of ROOT's core interpreter. I improved the correctness and reliability of `clang-repl` by fixing lots of bugs. I also added an initial implementation of the code undo feature. Here're some noticeable patches:

- https://reviews.llvm.org/D126781
- https://reviews.llvm.org/D126682
- https://reviews.llvm.org/D128782
- https://reviews.llvm.org/D130420
- https://reviews.llvm.org/D130831

It's not just about code!

During this summer, I have learned a lot about coding stuff and enhanced my knowledge of C++ and compiler. However, I think the most crucial thing I have learned has nothing to do with programming itself.

Due to the huge legacy codebase and the complexity of C++ semantics itself, the knowledge is dispersed between various people and teams. In order to solve the problems, instead of sitting at the desk spend all day coding and debugging, we need to look for the right communication channels to extract the knowledge and turn it into an acceptable solution.

So remember human beings are collective animals and it's always better to work with a team!

Acknowledgment

Here I would like to thank everyone that help me with my GSoC project. Especially Vassil, he's not only an experienced programmer, but also a patient teacher and friend. He always gives me good ideas when I ran into some weird bugs, and provides useful suggestions for my career. I should also thanks David, who patiently helped me with the crazy CMSSW build infrastructure.