HEP Software Foundation

Optimize ROOT use of modules for large codebases

Mentors: Vassil Vassilev
Alexander Penev
David Lange

About Me

Name	**Jun Zhang**
Email	**jun@junz.org (Primary) nailuo2002@gmail.com**
Discord	**junaire#4703**
Github	**https://github.com/junaire**

Blog	**https://www.junz.org/**
Time Zone	**CST(UTC+8000)**

Education & Background

I’m a 2nd year undergraduate from Anhui Normal University (China) and I major in software engineering. I have a strong passion for open source, and I’m always eager to participate in it.
Last year, I applied for OSPP(Summer 2021 of Open Source Promotion Plan) [1]and worked in a community that focused on cloud storage (BeyondStorage[2]). My main task is to add a new service for it, and all my work can be found here.
I also try to get involved in the LLVM community, mainly focused on contributing to the Clang and Clang tools.

Here are some of my patches that are worth mentioning:

Title	Link	Status
[Clang] Add __builtin_elementwise_* and __builtin_reduce_*	https://reviews.llvm.org/D114688 https://reviews.llvm.org/D115231 https://reviews.llvm.org/D115429 https://reviews.llvm.org/D116161 https://reviews.llvm.org/D116736	**Merged**
[Clang-tidy] Check the existence of ElaboratedType's qualifiers	https://reviews.llvm.org/D119949	**Merged**
[Clang][Sema] Prohibit statement expression in the default argument	https://reviews.llvm.org/D119609	**Merged**
[Clang] Fix unknown type	https://reviews.llvm.org/D12	**Merged**

[1]https://summer.iscas.ac.cn/help/en
[2]https://beyondstorage.io

attributes diagnosed twice with [[]] spelling	3447	
[Clang][Sema] Fix invalid redefinition error in if/switch/for statement	https://reviews.llvm.org/D123840	**Merged**

You can find all my contributions here as well.

For the programming languages, I have intermediate-level knowledge of C++ and Python3.

Introduction

ROOT is a data analysis framework that is broadly used in and outside of High Energy Physics (HEP). Since HEP software frameworks always deal with massive data, the performance of ROOT becomes a critical aspect of our attention. ROOT is written in C++ and built upon the LLVM technology, and a well-known issue about C++ is its inefficient header parsing strategy. This can be worse because Cling (ROOT's C++ interpreter) needs to parse source code at runtime. Thanks to LLVM, there's already a C++ module implementation in the Clang frontend, and it works as an API for ROOT.

But the current C++ module integration also has its own limitations. For example, one source of performance loss is the need for symbol lookups across a very large set of modules.
If we want to lookup an identifier like `X`, and `X` is defined in namespace `edm`, like code below:

```
namespace edm {
// tons of unrelated code
int X = 42;
// another tons of unrelated code
}
```

Then ROOT will pull all modules defining the same namespace, which is totally unnecessary and expensive, and this proposal intends to optimize this corner of ROOT.

Implementation

To achieve the best performance, ROOT loads prebuilt modules using a global module index (GMI). A global module index is a data structure that can map the identifier and the module that contains the corresponding definition efficiently. The related logic is in `core/metacling/src/TCling.cxx`

```
static GlobalModuleIndex *loadGlobalModuleIndex(cling::Interpreter &interp) {
  // ...
  clang::GlobalModuleIndex::UserDefinedInterestingIDs IDs;

  struct DefinitionFinder : public RecursiveASTVisitor<DefinitionFinder> {
    DefinitionFinder(clang::GlobalModuleIndex::UserDefinedInterestingIDs &IDs,
clang::TranslationUnitDecl *TU)
        : DefinitionIDs(IDs) {
      TraverseDecl(TU);
    }
    bool VisitNamedDecl(NamedDecl *ND) {
      if (...)
        return true;
      // ...
      else if (NamespaceDecl *NSD = llvm::dyn_cast<NamespaceDecl>(ND)) {
        Register(NSD, /*AddSingleEntry=*/false);
      }
      return true;
    }

  private:
    clang::GlobalModuleIndex::UserDefinedInterestingIDs &DefinitionIDs;
    void Register(const NamedDecl *ND, bool AddSingleEntry = true) {
      // ...
      DefinitionIDs[ND->getName()].push_back(OwningModule->getASTFile());
    }
  };
  DefinitionFinder defFinder(IDs,
CI.getASTContext().getTranslationUnitDecl());

  llvm::cantFail(GlobalModuleIndex::writeIndex(
      CI.getFileManager(), CI.getPCHContainerReader(), ModuleIndexPath, &IDs));
  return GlobalIndex;
}
```

We can observe that when we traverse the whole AST and find a `NamedDecl` is a `NamespaceDecl`, we will ‘register’ it in the `UserDefinedInterestingIDs`, which is a `StringMap` that key is the name of the identifier and value is a vector that contains all modules it appears. Then, the `GlobalModuleIndexBuilder` will write the index to the GMI.

The lookup logic can be found in `core/metacling/src/TClingCallbacks.cxx`:

```
bool TClingCallbacks::findInGlobalModuleIndex(
    DeclarationName Name, bool loadFirstMatchOnly /*=true*/) {
  // ...
  if (Index->lookupIdentifier(Name.getAsString(), FoundModules)) {
    for (llvm::StringRef FileName : FoundModules) {
      // ...
      m_Interpreter->loadModule(ModuleName);
      m_LoadedModuleFiles[FileName] = Name;
    }
    return true;
  }
  return false;
}
```

So what the code above does is fairly clear: we find all modules that contain the identifier, and load them on demand. Now we can see the problem: if we type something like `A::B`, we will look up identifier `A` first, then load up whole modules unnecessarily.

So what we want to do is simple: **Ignore identifiers that declare a namespace**.

To ignore identifiers that define a namespace, we have to track them when we write index to the GMI. First, we can add a new overload for `GlobalModuleIndex::writeIndex` or add a default argument to the existing one, then by tracking the identifier's name and its `DeclKind`, we will have the knowledge about how to deal with them later on.

When it comes to `GlobalModuleIndex::lookupIdentifier`, we will be able to know the `DeclKind` of the identifiers. So if we are looking up an identifier that defines a namespace, we can stop here and ignore it.

So how should we store the identifier and its `DeclKind` mapping?

The current implementation of `GlobalModuleIndex` is a persistent hash table, and it maps the Identifier name ⇔ Module. In this case, we need to extend it to {identifier, `DeclKind`} ⇔ Module to solve our problem.

Deliverables

- Extend the `GlobalModuleIndex` structure, making it capable of storing the identifier's name and `DeclKind` mapping.
- Add a new overload or a default argument to the `GlobalModuleIndex::writeIndex`
- Add a new overload to the `GlobalModuleIndex::lookupIdentifier`, so we can explicitly ignore some `DeclKind`, here is `NamespaceDecl`.
- Track down the test failures of CMSSW and check if our proposed implementation works.

- Profile ROOT before and after the optimization, compare their benchmark, and present the work.

Timeline

Community Bonding period

- Read the documentation, play with ROOT and get familiar with its architecture.
- Learn the basic usage of LLVM’s bitstream.
- Contact with mentors, discuss the details about the new design of the `GlobalModuleIndex`

Week 1 (14th June - 21st June)

- Set up my working environment, get familiar with the existing implementation of the `GlobalModuleIndex`
- Set up performance tools such as perf, profile the performance of existing ROOT modules.

  Deliverable: Working environment and a basic report on my local performance tests.

Week 2 (21st June - 28th June)

- Debug and understand the workflow of the compiler and the relevant callbacks (in `TClingCallbacks.cxx`) when processing a namespace partition.
- Check the performance impact when disabling forward namespace declaration in module loading.
- Start working on a more advanced implementation in the `GlobalModuleIndex`
  Deliverable: send an initial patch of the new implementation of `GlobalModuleIndex`

Week 3 (28th June - 9th July)

I have university commitments during that period. I will still be in touch with my mentors, participate if possible in meetings and reply to emails.

Week 4 (9th July - 16th July)

- Enhance `GlobalModuleIndex::writeIndex`, store the {identifier, `DeclKind`} ⇔ {Module1, Module2, … ModuleN} mapping.

  Deliverable: deserialize all declarations from a given module

Week 5 (16th July - 23nd July)

- Measure the memory pressure when deserializing declarations from all modules when writing the GMI. If it turns out ROOT consumes too much memory, we can use some techniques like storing the extra information in the low bits to optimize it.

- **Deliverable:** A preliminary report on the memory pressure of the new design.

Week 6 (23nd July - 30th August)

- Use this week as a buffer since the last 2 weeks will be our major of work.

Week 7 (30th July - 6 August)

- Polish the implementation or investigate alternative selective reading strategies if the memory use is too high.

  Deliverable: Now we should be able to ignore loading modules based on a namespace partition lookup.

Week 8 (6th August - 13th August)

- Add a new overload to the `GlobalModuleIndex::lookupIdentifier`, ignore identifiers that define a namespace.
- Deploy the new implementation and gather feedback from users such as the CMSSW software package.

Week 9 (13th August - 27th August)

- Profile ROOT, examine the performance impact.

  Deliverable: A more advanced report about the performance impact of our new `GlobalModulendex` design. The report will include information about the peak memory usage with (`/usr/bin/time -v`); speed using Linux's kernel performance profiler (perf) and based on internal clang/ROOT infrastructure (eg, `ROOTDEBUG=3`).

Week 10 (3rd September - 10th September)

- Improve documentation.
- Add more tests and benchmarks.
- Consider upstreaming on the LLVM mainline of the developed patches.

  Deliverable: Open LLVM reviews.

Week 11 (27th August - 3rd September)

- Use these a week as a buffer week to catch up in case something took longer than expected.

Week 12 (3rd September - 10th September)

- Write a blog post (in English) to summarize this wonderful experience, including what I have learned and what mistakes have I made.
- Prepare a presentation to the mentors, submit a report to the ROOT repo, introduce the new design to the audience, and the performance improvement.

Extended Weeks

Thanks to Google, this year GSoC allows us to have extra weeks if we fail to complete the project. In our case, if the proposed implementation can’t work or result in a poor benchmark, we can use these weeks to figure out and fix it.

Availability

My summer vacation is expected to start in early July, and I will be back in school on September 1st. So most of my work will be done during the summer vacation. This summer I will be focusing on GSoC so I’m able to work about 45 hours a week. However, when I'm in school (before week 4 and the last week), I may only be able to only spend 25 hours a week due to my own busy school schedule.
If I can't do the work I'm supposed to do because of my studies, I can increase my workload for the summer, considering that I have a relatively free vacation.

PS: The schedule may change a little bit due to covid-19.

For communication, you can always contact me via discord, as long as I’m not sleeping.

Post GSoC

I really love the idea of open-source, where people gather together and work for the same goal. In the community, we do not distinguish between race, color, religion, and all prejudices, and communicate only through code. After the GSoC, I would like to write a blog post to summarize this experience and continue devoting myself to the community.