

Improving performance of C++ modules in Clang

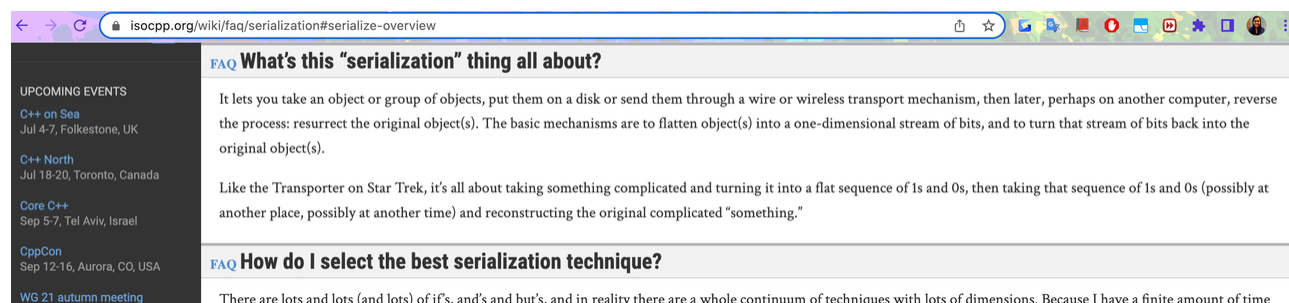
Problem Statement

The C++ modules technology aims to provide a scalable compilation model for the C++ language. The C++ Modules technology in Clang provides an io-efficient, on-disk representation capable to reduce build times and peak memory usage. The internal compiler state such as the **abstract syntax tree (AST) is stored on disk and lazily loaded on demand**. C++ Modules improve the memory footprint for interpreted C++ through the Cling C++ interpreter developed by CERN and the compiler research group at Princeton. The current implementation is pretty good at making most operations on demand.

However in a few cases, **we eagerly load pieces of the AST**, for example **at module import time** and upon **selecting a suitable template specialization**. When selecting the template specialization **we load all template specializations from the module files just to find out they are not suitable**. There is a patch that partially solves this issue by introducing a template argument hash and use it to look up the candidates without deserializing them. However, **the data structure it uses to store the hashes leads to quadratic search which is inefficient when the number of modules becomes sufficiently large**.

Serialization

Serialization is the process of writing or reading an object to or from a persistent storage medium such as a disk file.



The screenshot shows a web browser window with the address bar displaying `isocpp.org/wiki/faq/serialization#serialize-overview`. The page content is divided into two main sections. The first section, titled "FAQ What's this 'serialization' thing all about?", explains that serialization involves taking an object or group of objects, putting them on a disk or sending them through a transport mechanism, and then later reversing the process to resurrect the original object(s). It also uses the analogy of the Transporter on Star Trek. The second section, titled "FAQ How do I select the best serialization technique?", states that there are many techniques with various dimensions, and the choice depends on the specific requirements.

UPCOMING EVENTS

- C++ on Sea
Jul 4-7, Folkestone, UK
- C++ North
Jul 18-20, Toronto, Canada
- Core C++
Sep 5-7, Tel Aviv, Israel
- CppCon
Sep 12-16, Aurora, CO, USA
- WG 21 autumn meeting

FAQ What's this "serialization" thing all about?

It lets you take an object or group of objects, put them on a disk or send them through a wire or wireless transport mechanism, then later, perhaps on another computer, reverse the process: resurrect the original object(s). The basic mechanisms are to flatten object(s) into a one-dimensional stream of bits, and to turn that stream of bits back into the original object(s).

Like the Transporter on Star Trek, it's all about taking something complicated and turning it into a flat sequence of 1s and 0s, then taking that sequence of 1s and 0s (possibly at another place, possibly at another time) and reconstructing the original complicated "something."

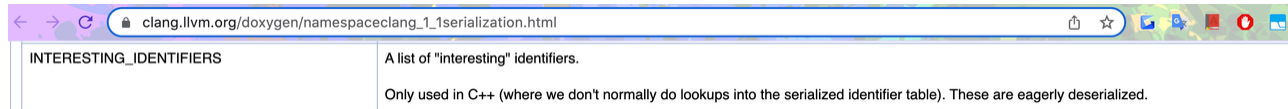
FAQ How do I select the best serialization technique?

There are lots and lots (and lots) of if's, and's and but's, and in reality there are a whole continuum of techniques with lots of dimensions. Because I have a finite amount of time

Deserialization

The byte stream, once created, also can be streamed across a communication link to a remote receiving end. The reverse of serialization is called deserialization, where the data in the byte stream is used to reconstruct it to its original object form.

Eager Deserialization



Example when eager deserialization cannot be avoided: until c++20 we could lazily deserialize the vtable information but due to `constexpr virtual` in c++20 we cannot anymore.

In Clang

Serialization and Deserialization

```
clang/include/clang/Serialization/ASTDeserializationListener.h
```

```

#include "clang/Basic/IdentifierTable.h"
#include "clang/Serialization/ASTBitCodes.h"

namespace clang {

class Decl;
class ASTReader;
class QualType;
class MacroDefinitionRecord;
class MacroInfo;
class Module;
class SourceLocation;

class ASTDeserializationListener {
public:
    virtual ~ASTDeserializationListener();

    /// The ASTReader was initialized.
    virtual void ReaderInitialized(ASTReader *Reader) { }

    /// An identifier was deserialized from the AST file.
    virtual void IdentifierRead(serialization::IdentID ID,
                               IdentifierInfo *II) { }

    /// A macro was read from the AST file.
    virtual void MacroRead(serialization::MacroID ID, MacroInfo *MI) { }

    /// A type was deserialized from the AST file. The ID here has the
    ///     qualifier bits already removed, and T is guaranteed to be locally
    ///     unqualified.
    virtual void TypeRead(serialization::TypeIdx Idx, QualType T) { }

    /// A decl was deserialized from the AST file.
    virtual void DeclRead(serialization::DeclID ID, const Decl *D) { }

    /// A selector was read from the AST file.
    virtual void SelectorRead(serialization::SelectorID iD, Selector Sel) {}

    /// A macro definition was read from the AST file.
    virtual void MacroDefinitionRead(serialization::PreprocessedEntityID,
                                     MacroDefinitionRecord *MD) {}

    /// A module definition was read from the AST file.
    virtual void ModuleRead(serialization::SubmoduleID ID, Module *Mod) {}

    /// A module import was read from the AST file.
    virtual void ModuleImportRead(serialization::SubmoduleID ID,
                                  SourceLocation ImportLoc) {}
};

```

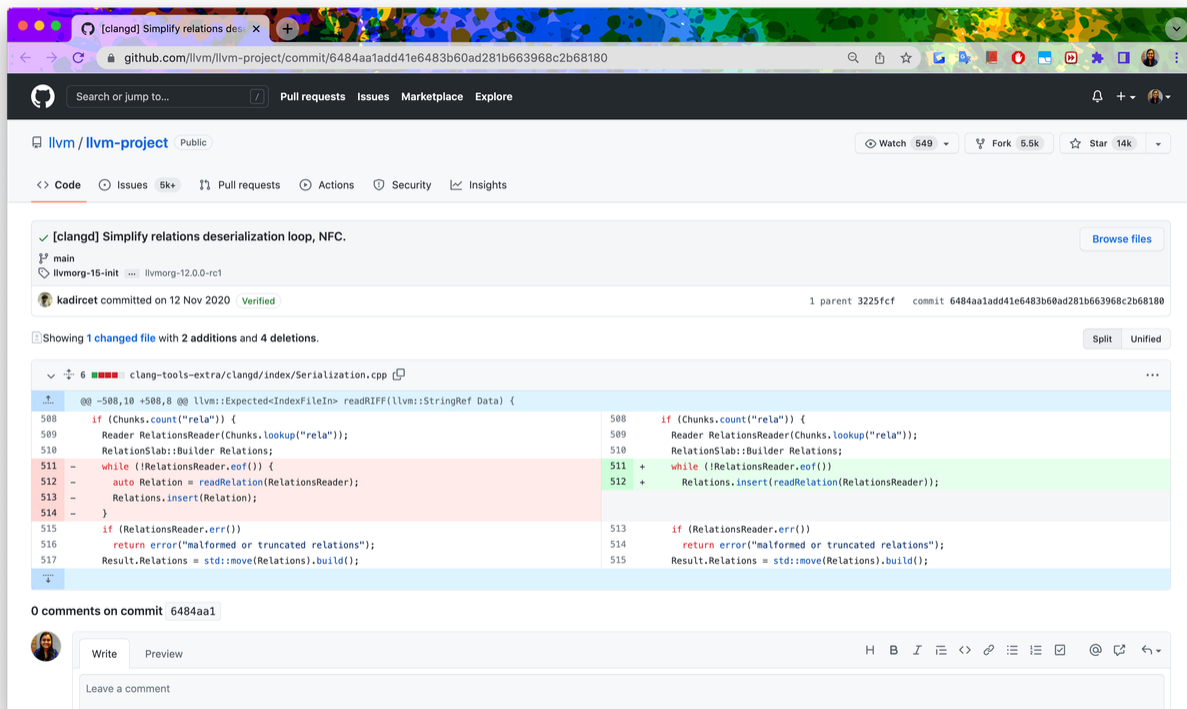
```

void ASTWriter::ModuleRead(serialization::SubmoduleID ID, Module *Mod) {
    assert(SubmoduleIDs.find(Mod) == SubmoduleIDs.end());
    SubmoduleIDs[Mod] = ID;
}

```

}

Simple code to understand deserialization



Eager Deserialization

Module import time

<https://github.com/llvm/llvm-project/commit/c52efa7d4011a48ea91b353f2cbc40a359d19571>

[modules] Don't eagerly deserialize so many ImportDecls. CodeGen basi...

...cally ignores ImportDecls imported from modules, so only eagerly deserialize the ones from a PCH / preamble.

llvm-svn: 245406

main
llvmorg-15-init ... 2020.06-alpha

zygoid committed on 19 Aug 2015

1 parent 72be1c1 commit c52efa7d4011a48ea91b353f2cbc40a359d19571

Showing 2 changed files with 13 additions and 11 deletions.

SplitUnified

clang/lib/CodeGen/CodeGenModule.cpp

@@ -3363,11 +3363,8 @@ void CodeGenModule::EmitTopLevelDecl(Decl *D) {	
3363 auto *Import = cast<ImportDecl>(D);	3363 auto *Import = cast<ImportDecl>(D);
3364	3364
3365 // Ignore import declarations that come from imported modules.	3365 // Ignore import declarations that come from imported modules.
3366 - if (clang::Module *Owner = Import->getImportedOwningModule()) {	3366 + if (Import->getImportedOwningModule())
3367 - if (getLangOpts().CurrentModule.empty()	3367 + break;
3368 - Owner->getTopLevelModule()->Name == getLangOpts().CurrentModule	
3369 - break;	
3370 - }	
3371 if (CGDebugInfo *DI = getModuleDebugInfo())	3368 if (CGDebugInfo *DI = getModuleDebugInfo())
3372 DI->EmitImportDecl(*Import);	3369 DI->EmitImportDecl(*Import);
3373	3370
....	

clang/lib/Serialization/ASTWriterDecl.cpp

@@ -1994,14 +1994,19 @@ void ASTWriter::WriteDeclAbbrevs() {	
1994 /// clients to use a separate API call to "realize" the decl. This should be	1994 /// clients to use a separate API call to "realize" the decl. This should be

Upon selecting a suitable template specialization

When selecting the template specialization we load all template specializations from the module files just to find out they are not suitable.

Don't eagerly deserialize every templated function (and every static ...

...data

member inside a class template) when loading a PCH file or module.

llvm-svn: 178496

main
studio-1.4 ... 2020.06-alpha

zygoid committed on 2 Apr 2013

1 parent 3435327 commit 5205a8cfd815e4e71ccd3f067080c11c8980c399

Showing 3 changed files with 20 additions and 2 deletions.

SplitUnified

clang/lib/AST/ASTContext.cpp

@@ -7673,7 +7673,15 @@ bool ASTContext::DeclMustBeEmitted(const Decl *D) {	
7673 if (const VarDecl *VD = dyn_cast<VarDecl>(D)) {	7673 if (const VarDecl *VD = dyn_cast<VarDecl>(D)) {
7674 if (!VD->isFileVarDecl())	7674 if (!VD->isFileVarDecl())
7675 return false;	7675 return false;
7676 - } else if (!isa<FunctionDecl>(D))	7676 + } else if (const FunctionDecl *FD = dyn_cast<FunctionDecl>(D)) {
	7677 + // We never need to emit an uninstantiated function template.
	7678 + if (FD->getTemplatedKind() == FunctionDecl::TK_FunctionTemplate)
	7679 + return false;
	7680 + } else
	7681 + return false;
	7682 +
	7683 + // If this is a member of a class template, we do not need to emit it.
	7684 + if (D->getDeclContext()->isDependentContext())
7677 return false;	7685 return false;
7678	7686
7679 // Weak references don't produce any output by themselves.	7687 // Weak references don't produce any output by themselves.
....	

clang/test/PCH/cxx-templates.cpp

With lazy deserialization, builtins are loaded on-demand, and unused builtins are never loaded into the Isolate. Lazy deserialization comes with memory savings.

Existing (using print statements)

<https://github.com/llvm/llvm-project/blob/main/clang/include/clang/Serialization/ASTBitCodes.h#L484-L492>

EAGERLY_DESERIALIZED_DECLS

```
github.com/llvm/llvm-project/blob/main/clang/include/clang/Serialization/ASTBitCodes.h#L484-L492
... 484    /// Record code for the array of eagerly deserialized decls.
    485    ///
    486    /// The AST file contains a list of all of the declarations that should be
    487    /// eagerly deserialized present within the parsed headers, stored as an
    488    /// array of declaration IDs. These declarations will be
    489    /// reported to the AST consumer after the AST file has been
    490    /// read, since their presence can affect the semantics of the
    491    /// program (e.g., for code generation).
    492    EAGERLY_DESERIALIZED_DECLS = 6,
    493
```

1. <https://github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L3259-L3264>

```
github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L3259-L3264
... 3258
    3259    case EAGERLY_DESERIALIZED_DECLS:
    3260        // FIXME: Skip reading this record if our ASTConsumer doesn't care
    3261        // about "interesting" decls (for instance, if we're building a module).
    3262        for (unsigned I = 0, N = Record.size(); I != N; ++I)
    3263            EagerlyDeserializedDecls.push_back(getGlobalDeclID(F, Record[I]));
    3264        break;
    3265
    3266    case MODULAR_CODEGEN_DECLS:
    3267        // FIXME: Skip reading this record if our ASTConsumer doesn't care about
    3268        // them (ie: if we're not codegenerating this module).
    3269        if (F.Kind == MK_MainFile ||
```

2. <https://github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L7486>

github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L7486

```
7486     if (!DeclsLoaded[Index]) {
7487         ReadDeclRecord(ID);
7488         if (DeserializationListener)
7489             DeserializationListener->DeclRead(ID, DeclsLoaded[Index]);
7490     }
7491
7492     return DeclsLoaded[Index];
7493 }
7494
```

3. <https://github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L1573-L1604>

Preallocated source locations for modules which are not loaded. There was some plan to reduce this but didn't go anywhere.

github.com/llvm/llvm-project/blob/main/clang/lib/Serialization/ASTReader.cpp#L1573-L1604

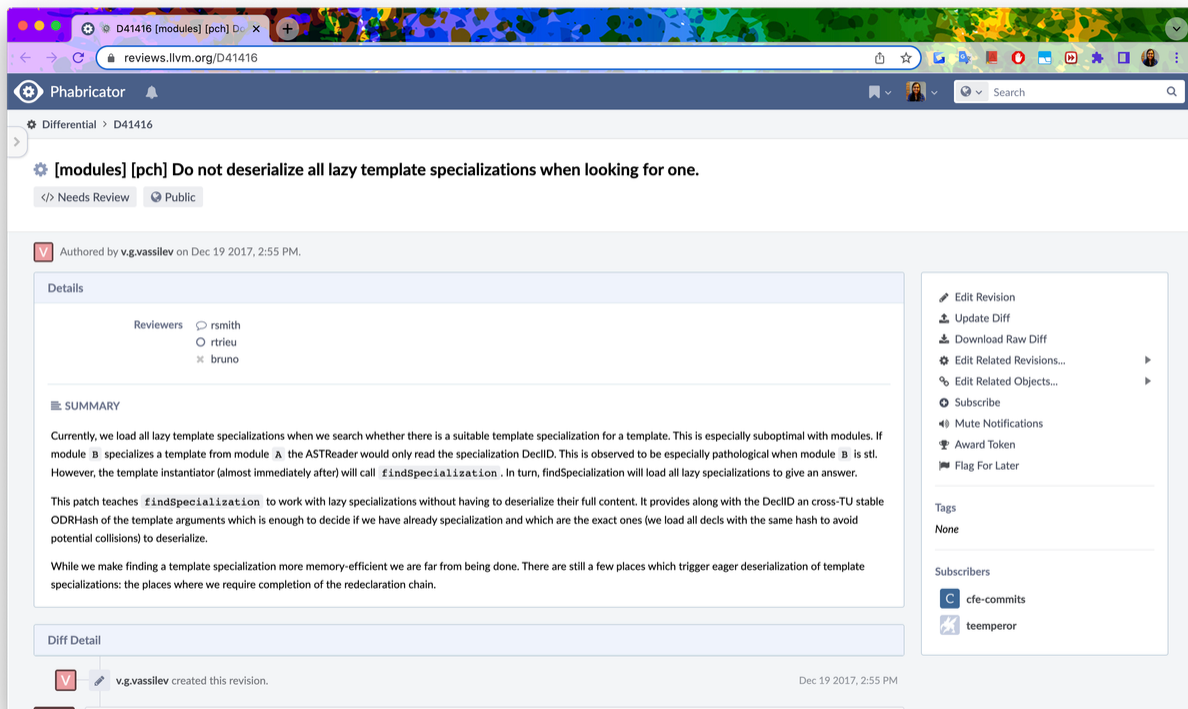
```
1573     case SM_SLOC_BUFFER_ENTRY: {
1574         const char *Name = Blob.data();
1575         unsigned Offset = Record[0];
1576         SrcMgr::CharacteristicKind
1577             FileCharacter = (SrcMgr::CharacteristicKind)Record[2];
1578         SourceLocation IncludeLoc = ReadSourceLocation(*F, Record[1]);
1579         if (IncludeLoc.isInvalid() && F->isModule()) {
1580             IncludeLoc = getImportLocation(F);
1581         }
1582
1583         auto Buffer = ReadBuffer(SLocEntryCursor, Name);
1584         if (!Buffer)
1585             return true;
1586         SourceMgr.createFileID(std::move(Buffer), FileCharacter, ID,
1587                               BaseOffset + Offset, IncludeLoc);
1588         break;
1589     }
1590
1591     case SM_SLOC_EXPANSION_ENTRY: {
1592         LocSeq::State Seq;
1593         SourceLocation SpellingLoc = ReadSourceLocation(*F, Record[1], Seq);
1594         SourceLocation ExpansionBegin = ReadSourceLocation(*F, Record[2], Seq);
1595         SourceLocation ExpansionEnd = ReadSourceLocation(*F, Record[3], Seq);
1596         SourceMgr.createExpansionLoc(SpellingLoc, ExpansionBegin, ExpansionEnd,
1597                                     Record[5], Record[4], ID,
1598                                     BaseOffset + Record[0]);
1599         break;
1600     }
1601 }
1602
1603 return false;
1604 }
```

4. Another preloading: <https://github.com/root-project/root/blob/master/interpreter/llvm/src/tools/clang/lib/Serialization/ASTReader.cpp#L3176>

```
github.com/root-project/root/blob/master/interpreter/llvm/src/tools/clang/lib/Serialization/ASTReader.cpp#L3176
... 3176 F.PreloadIdentifierOffsets.assign(Record.begin(), Record.end());
    3177 break;
    3178
    3179 case EAGERLY_DESERIALIZED_DECLS:
    3180     // FIXME: Skip reading this record if our ASTConsumer doesn't care
    3181     // about "interesting" decls (for instance, if we're building a module).
    3182     for (unsigned I = 0, N = Record.size(); I != N; ++I)
    3183         EagerlyDeserializedDecls.push_back(getGlobalDeclID(F, Record[I]));
    3184     break;
    3185
```

Previous work

<https://reviews.llvm.org/D41416>



Partially solves this issue by introducing a template argument hash and use it to look up the candidates without deserializing them.

This way we managed to catch a few collisions in the ODRHash logic.

Check if we have already specialization and which are the exact ones (we load all decls with the same hash to avoid potential collisions) to deserialize.

Improvement/Optimization: the data structure it uses to store the hashes leads to quadratic search which is inefficient when the number of modules becomes sufficiently large.

Roadmap

Investigate and resolve eager deserialization where possible

1. Use the internal clang AST counters to file what is eagerly deserialize.
2. Add `printf` in `ASTReader::ReadDecl` and load a bunch of modules without using them. This ideally should be a nop. If that's not the case it has to be debugged and investigated further.

Rework the patch to use on-disk hash tables to avoid the quadratic search complexity

1. Move to using an on-disk hash table for template specialization lookup, at least for templates with large numbers of specializations
2. Currently when we hash a tag type the visitor calls `ODRHash::AddDecl` which mostly relies on the decl name give distinct hash value. The types coming from template specializations have very similar properties (including decl names). For those we need to provide more information in order to disambiguate them. This patch adds the template arguments for the template specialization decl corresponding to its type. We manage to reduce further the amount of deserializations from 1117 down to 451.



v.g.vassilev updated this revision to Diff 128802.

Jan 5 2018, 2:32 PM

Reduce further the deserializations from 451 to 449 by providing a more complete implementation of `ODRHash::AddTemplateArgument`.

Kudos @rtreu !

3. Stats:

- `types read` is down from 30% to 17%
- `declarations read` is down from 34% to 23%
- number of `ClassTemplateSpecializations` read has decreased by 30%,
- number of `CXXRecordDecls` read is down 25%
- total `ASTContext` memory usage is down by 12%

4. calculate hash

```
704
705 unsigned TemplateArgumentList::ComputeODRHash(ArrayRef<TemplateArgument> Args) {
706     ODRHash Hasher;
707     for (TemplateArgument TA : Args)
708         Hasher.AddTemplateArgument(TA);
709
710     return Hasher.CalculateHash();
711 }
712
713 FunctionTemplateSpecializationInfo <
```

5. Add template argument

```
// If this was a specialization we should take into account its template
// arguments. This helps to reduce collisions coming when visiting template
// specialization types (eg. when processing type template arguments).
ArrayRef<TemplateArgument> Args;
if (auto *CTSD = dyn_cast<ClassTemplateSpecializationDecl>(D))
    Args = CTSD->getTemplateArgs().asArray();
else if (auto *VTSD = dyn_cast<VarTemplateSpecializationDecl>(D))
    Args = VTSD->getTemplateArgs().asArray();
else if (auto *FD = dyn_cast<FunctionDecl>(D))
    if (FD->getTemplateSpecializationArgs())
        Args = FD->getTemplateSpecializationArgs()->asArray();

for (auto &TA : Args)
    AddTemplateArgument(TA);
```

6. ASTWriter.cpp

```

6996     auto *DC = cast<DeclContext>(DCDecl);
6997     SmallVector<Decl*, 8> Decls;
6998     FindExternalLexicalDecls(
6999         DC, [&](Decl::Kind K) { return K == D->getKind(); }, Decls);
7000     }
7001     }
7002     }
7003
7004     if (auto *CTSD = dyn_cast<ClassTemplateSpecializationDecl>(D))
7005         CTSD->getSpecializedTemplate()->LoadLazySpecializations();
7006     if (auto *VTSD = dyn_cast<VarTemplateSpecializationDecl>(D))
7007         VTSD->getSpecializedTemplate()->LoadLazySpecializations();
7008     if (auto *FD = dyn_cast<FunctionDecl>(D)) {
7009         if (auto *Template = FD->getPrimaryTemplate())
7010             Template->LoadLazySpecializations();
7011     }
7012     }
7013     }

```

```

6996     auto *DC = cast<DeclContext>(DCDecl);
6997     SmallVector<Decl*, 8> Decls;
6998     FindExternalLexicalDecls(
6999         DC, [&](Decl::Kind K) { return K == D->getKind(); }, Decls);
7000     }
7001     }
7002     }
7003
7004     RedefinableTemplateDecl *Template = nullptr;
7005     ArrayRef<TemplateArgument> Args;
7006     if (auto *CTSD = dyn_cast<ClassTemplateSpecializationDecl>(D)) {
7007         Template = CTSD->getSpecializedTemplate();
7008         Args = CTSD->getTemplateArgs().asArray();
7009     } else if (auto *VTSD = dyn_cast<VarTemplateSpecializationDecl>(D)) {
7010         Template = VTSD->getSpecializedTemplate();
7011         Args = VTSD->getTemplateArgs().asArray();
7012     } else if (auto *FD = dyn_cast<FunctionDecl>(D)) {
7013         if (auto *Tmpl = FD->getPrimaryTemplate()) {
7014             Template = Tmpl;
7015             Args = FD->getTemplateSpecializationArgs()->asArray();
7016         }
7017     }
7018
7019     if (Template)
7020         Template->loadLazySpecializationsImpl(Args);
7021     }

```

rsmith

Done

7. Added template specialisation info.

```

5125     Record.push_back(Kind);
5126
5127     switch (Kind) {
5128     case UPD_CXX_ADDED_IMPLICIT_MEMBER:
5129     case UPD_CXX_ADDED_TEMPLATE_SPECIALIZATION:
5130     case UPD_CXX_ADDED_ANONYMOUS_NAMESPACE:
5131         assert(Update.getDecl() && "no decl to add?");
5132         Record.push_back(GetDeclRef(Update.getDecl()));
5133         break;
5134
5135     case UPD_CXX_ADDED_FUNCTION_DEFINITION:
5136         break;
5137
5138     case UPD_CXX_POINT_OF_INSTANTIATION:
5139         // FIXME: Do we need to also save the template specialization kind here?
5140         Record.AddSourceLocation(Update.getLoc());
5141         break;
5142

```

```

5125     Record.push_back(Kind);
5126
5127     switch (Kind) {
5128     case UPD_CXX_ADDED_IMPLICIT_MEMBER:
5129
5130     case UPD_CXX_ADDED_ANONYMOUS_NAMESPACE:
5131         assert(Update.getDecl() && "no decl to add?");
5132         Record.push_back(GetDeclRef(Update.getDecl()));
5133         break;
5134
5135     case UPD_CXX_ADDED_TEMPLATE_SPECIALIZATION: {
5136         const Decl *Spec = Update.getDecl();
5137         assert(Spec && "no decl to add?");
5138         Record.push_back(GetDeclRef(Spec));
5139         ArrayRef<TemplateArgument> Args;
5140         if (auto *CTSD = dyn_cast<ClassTemplateSpecializationDecl>(Spec))
5141             Args = CTSD->getTemplateArgs().asArray();
5142         else if (auto *VTSD = dyn_cast<VarTemplateSpecializationDecl>(Spec))
5143             Args = VTSD->getTemplateArgs().asArray();
5144         else if (auto *FD = dyn_cast<FunctionDecl>(Spec))
5145             Args = FD->getTemplateSpecializationArgs()->asArray();
5146         assert(Args.size());
5147         Record.push_back(TemplateArgumentList::ComputeODRHash(Args));
5148         bool IsPartialSpecialization
5149             = isa<ClassTemplatePartialSpecializationDecl>(Spec) ||
5150               isa<VarTemplatePartialSpecializationDecl>(Spec);
5151         Record.push_back(IsPartialSpecialization);
5152         break;
5153
5154     case UPD_CXX_ADDED_FUNCTION_DEFINITION:
5155         break;
5156
5157     case UPD_CXX_POINT_OF_INSTANTIATION:
5158         // FIXME: Do we need to also save the template specialization kind here?
5159         Record.AddSourceLocation(Update.getLoc());
5160         break;

```

8. <https://github.com/root-project/root/blob/master/interpreter/llvm/src/tools/clang/lib/Serialization/ASTReader.cpp#L3134-L3145>

```
github.com/root-project/root/blob/master/interpreter/llvm/src/tools/clang/lib/Serialization/ASTReader.cpp#L3134-L3145
... 3134     case IDENTIFIER_TABLE:
3135         F.IdentifierTableData = Blob.data();
3136         if (Record[0]) {
3137             F.IdentifierLookupTable = ASTIdentifierLookupTable::Create(
3138                 (const unsigned char *)F.IdentifierTableData + Record[0],
3139                 (const unsigned char *)F.IdentifierTableData + sizeof(uint32_t),
3140                 (const unsigned char *)F.IdentifierTableData,
3141                 ASTIdentifierLookupTrait(*this, F));
3142
3143             PP.getIdentifierTable().setExternalIdentifierLookup(this);
3144         }
3145         break;
3146
```

Read a blob of identifiers from a module file and then put that blob into that table which is of type `llvm::OnDiskIterableChainedHashTable`.

Measure performance improvements

Size — `du -sh *pcm`

sort largest to smallest measure of file space amount recursively stored in directory

Memory Consumption — `/usr/bin/time -v root.exe -l -b -q tutorials/hsimple.C`

Compared against eager deserialization, reduce heap size.

Use the internal performance counters in clang - <https://godbolt.org/z/s61fxoYPs>



Internal performance counters:

*** AST Context Stats:

25662 types total.

5 Decayed types, 48 each (240 bytes)

133 ConstantArray types, 56 each (7448 bytes)

21 DependentSizedArray types, 64 each (1344 bytes)

19 IncompleteArray types, 40 each (760 bytes)

62 Builtin types, 24 each (1488 bytes)

103 Decltype types, 40 each (4120 bytes)

18 Auto types, 48 each (864 bytes)

969 DependentName types, 48 each (46512 bytes)

43 DependentTemplateSpecialization types, 48 each (2064 bytes)

736 Elaborated types, 48 each (35328 bytes)

6419 FunctionProto types, 40 each (256760 bytes)

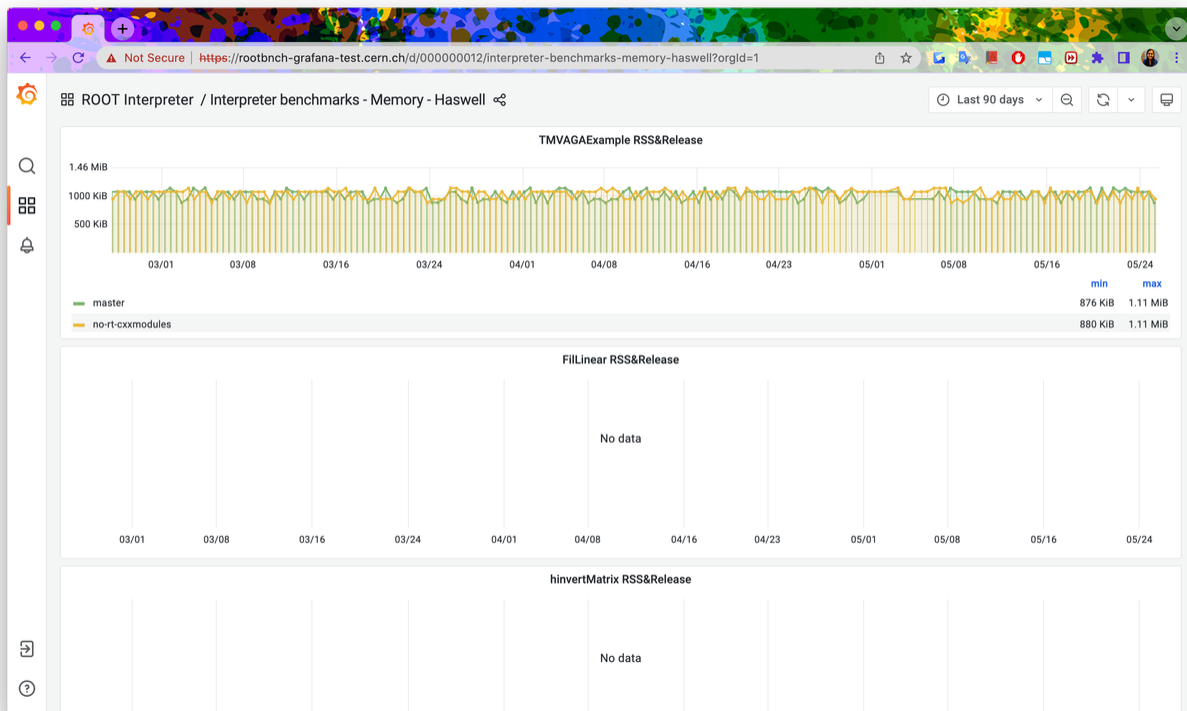
645 InjectedClassName types, 40 each (25800 bytes)

```
76 MemberPointer types, 48 each (3648 bytes)
148 PackExpansion types, 40 each (5920 bytes)
98 Paren types, 40 each (3920 bytes)
1861 Pointer types, 40 each (74440 bytes)
1505 LValueReference types, 40 each (60200 bytes)
324 RValueReference types, 40 each (12960 bytes)
1015 SubstTemplateTypeParm types, 40 each (40600 bytes)
87 Enum types, 32 each (2784 bytes)
716 Record types, 32 each (22912 bytes)
6815 TemplateSpecialization types, 40 each (272600 bytes)
2935 TemplateTypeParm types, 40 each (117400 bytes)
32 TypeOfExpr types, 32 each (1024 bytes)
869 Typedef types, 32 each (27808 bytes)
1 UnaryTransform types, 48 each (48 bytes)
7 Using types, 40 each (280 bytes)
Total bytes = 1029272
31/518 implicit default constructors created
98/591 implicit copy constructors created
54/543 implicit move constructors created
34/595 implicit copy assignment operators created
7/543 implicit move assignment operators created
43/544 implicit destructors created

Number of memory regions: 513
Bytes used: 7701107
Bytes allocated: 7929856
Bytes wasted: 228749 (includes alignment, etc)
```

Reduced memory consumption — ask Google to run the reimplementations of D41416 on their builds

<https://rootbnch-grafana-test.cern.ch/dashboards>



Build ROOT with `-Druntime_cxxmodules=On` on Windows

How to model the partial template specializations

Allows customizing class and variable templates for a given category of template arguments.

Examples of partial specializations in the standard library include `std::unique_ptr`, which has a partial specialization for array types.

example: from https://en.cppreference.com/w/cpp/language/partial_specialization

When a class or variable template is instantiated, and there are partial specializations available, the compiler has to decide if the primary template is going to be used or one of its partial specializations.

1) If only one specialization matches the template arguments, that specialization is used

2) If more than one specialization matches, partial order rules are used to determine which specialization is more specialized. The most specialized specialization is used, if it is unique (if it is not unique, the program cannot be compiled)

3) If no specializations match, the primary template is used

- the first function template has the same template parameters as the first partial specialization and has just one function parameter, whose type is a class template specialization with all the template arguments from the first partial specialization

- the second function template has the same template parameters as the second partial specialization and has just one function parameter whose type is a class template specialization with all the template arguments from the second partial specialization.

The function templates are then ranked as if for [function template overloading](#).