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

#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 ASTDeserializerListener {
public:
    virtual ~ASTDeserializerListener();

    // 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) {} 
};
Simple code to understand deserialization

Eager Deserialization

Module import time

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

```c
EAGERLY_DESERIALIZED_DECLS
```


```c
    case EAGERLY_DESERIALIZED_DECLS:
        // FIXME: Skip reading this record if our ASTConsumer doesn't care
        // about "interesting" decls (for instance, if we're building a module).
        for (unsigned I = 0, N = Record.size(); I != N; ++I)
            EagerlyDeserializedDecl.push_back(getGlobalDeclID(F, Record[I]));
        break;
```


```c
    case MODULAR_CODEGEN_DECLS:
        // FIXME: Skip reading this record if our ASTConsumer doesn't care about
        // them (ie: if we're not codegenerating this module).
        if (F.Kind == MK_MainFile ||
```
Preallocated source locations for modules which are not loaded. There was some plan to reduce this but didn't go anywhere.

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.
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 **CXXRecordDefs** read is down 25%
   - total **ASTContext** memory usage is down by 12%

4. calculate hash

```cpp
//143
unsigned TemplateArgumentList::ComputeODRHash(ArrayRef<TemplateArgument> Args) {
  ODRHash Hasher;
  for (TemplateArgument TA : Args)
    Hasher.AddTemplateArgument(TA);
  return Hasher.CalculateHash();
}
```

5. Add template argument

```cpp
// If this was a specialization we should take into account its template
// arguments. This helps to reduce collisions coming when visiting template
// specialization types (e.g. 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
7. Added template specialisation info.

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](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)
<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemberPointer types</td>
<td>76</td>
<td>(3648 bytes)</td>
</tr>
<tr>
<td>PackExpansion types</td>
<td>148</td>
<td>(5920 bytes)</td>
</tr>
<tr>
<td>Paren types</td>
<td>98</td>
<td>(3920 bytes)</td>
</tr>
<tr>
<td>Pointer types</td>
<td>1861</td>
<td>(74440 bytes)</td>
</tr>
<tr>
<td>LValueReference types</td>
<td>1505</td>
<td>(60200 bytes)</td>
</tr>
<tr>
<td>RValueReference types</td>
<td>324</td>
<td>(12960 bytes)</td>
</tr>
<tr>
<td>SubstTemplateTypeParm types</td>
<td>1015</td>
<td>(40600 bytes)</td>
</tr>
<tr>
<td>Enum types</td>
<td>87</td>
<td>(2784 bytes)</td>
</tr>
<tr>
<td>Record types</td>
<td>716</td>
<td>(22912 bytes)</td>
</tr>
<tr>
<td>TemplateSpecialization types</td>
<td>6815</td>
<td>(272600 bytes)</td>
</tr>
<tr>
<td>TemplateTypeParm types</td>
<td>2935</td>
<td>(117400 bytes)</td>
</tr>
<tr>
<td>.TypeOfExpr types</td>
<td>32</td>
<td>(1024 bytes)</td>
</tr>
<tr>
<td>Typedef types</td>
<td>869</td>
<td>(27808 bytes)</td>
</tr>
<tr>
<td>UnaryTransform types</td>
<td>1</td>
<td>(48 bytes)</td>
</tr>
<tr>
<td>Using types</td>
<td>7</td>
<td>(280 bytes)</td>
</tr>
</tbody>
</table>

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 reimplementation of D41416 on their builds
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.