Debug Information
From Metadata to Modules

Adrian Prantl
Apple

Duncan Exon Smith
Apple
What **is** Debug Information?

- provides a mapping from *source code* → *binary program*
- on disk: as DWARF, a highly compressed format
- in LLVM: as metadata (pre-finalized DWARF)

![DWARF debug information for clang r250459, RelWithDebInfo+Assertions](image)
Debug Info, Scalability, and LTO

• volume of debug info limits scalability of the compiler, particularly when using LTO

• we attacked this problem from two sides:
  • LLVM: efficient new Metadata representation
  • Clang: emit less debug info with Module Debugging
LLVM: efficient new Metadata representation

- making Metadata lightweight: dropping use-lists and separating from Value
- specialized MDNodes: syntax, isa support, and memory footprint
- constructing Metadata graphs efficiently and distinct Metadata
- grab bag of other major LTO optimizations
Making Metadata lightweight

old class hierarchy

MDString  MDNode  Argument  User
How do operands work?
How do operands work?

intrusive storage for use-lists
How do operands work?

User operands are an array of Uses

User
- vtable
- type
- uselist
- flags
- operands

Use
- value
- next
- prev

MDString
MDNode
Argument
User
How do operands work?

Value Handles are second-class
How did operands work?

old MDNode operands were an array of ValueHandles
Separating Metadata from Value

Diagram showing the separation of metadata and value with nodes representing MDString and MDNode connected to Metadata and Value nodes, and Argument and User nodes connected to Value node.
Separating Metadata from Value

Metadata

MDString

MDNode
Metadata is lightweight

- no vtable
- no use-lists
- no Type pointer

Metadata base class has size of 1 pointer
Metadata is lightweight

new MDNode operands are 4x smaller
Specialized MDNodes for debug info
MDTuple: generic MDNode

old MDNode syntax

!1 = metadata ![metadata ![2, metadata !"string"]]

MDTuple syntax

!1 = ![!2, !"string"]

isa support

if (isa<MDTuple>(N)) { ... }
DILocation: syntax

\[
!1 = \text{metadata } \{ \text{i32 30, i32 7, metadata } !2, \text{ null} \}
\]

\[
!1 = \text{!DILocation(line: 30, column: 7, scope: !2)}
\]
if (DINode(N).isLocation()) { ... }

if (auto *N = dyn_cast<MDNode>(V))
  if ((N->getNumOperands() == 3 ||
       N->getNumOperands() == 4) &&
       isa<ConstantInt>(N->getOperand(0)) &&
       isa<ConstantInt>(N->getOperand(1)) &&
       DINode(N).isScope(N->getOperand(2)) { ... }

if (isa<DILocation>(N)) { ... }

if (isLocation()):
DILocation: memory footprint

Metadata

MDString

MDNode

DILocation

16-bit column

32-bit line

md-flags
node-flags
context

scope

inlinedAt

Op

Op

Metadata*

Metadata*
What about other Metadata graphs?

- we should have more primitives for generic Metadata
  - MDInt and MDFloat: skip ConstantInt and ConstantFloat
  - vectors, dictionaries and lists (when tuples don't fit)
- specialized nodes: syntax, isa support, and memory footprint
  - what makes a graph important and/or stable enough?
  - can we enable it for out-of-tree nodes?
Constructing Metadata graphs

• frontends (DIBuilder), bitcode deserialization, and lib/Linker build metadata graphs

• need temporary nodes for forward references

• need use-lists (and RAUW support) to replace temporary nodes
  • Metadata use-lists are second-class
  • how can we limit exposure to use-lists?
Temporary storage for explicit use-lists

- largely unoptimized
- uses side storage
- dropped automatically, except uniquing cycles
Constructing a graph

\[ !0 = !\{!1\} \]
\[ !1 = !\{!2\} \]
\[ !2 = !\{\} \]

how can we build this graph?
Constructing a graph, top-down

!0 = {!1}
!1 = {!2}
!2 = {}

create temporary node for !1
Constructing a graph, top-down

0 = \{1\}
1 = \{2\}
2 = \{\}

create (unresolved) node for 0
Constructing a graph, top-down

!0 = {!1}
!1 = {!2}
!2 = {}
Constructing a graph, top-down

\[ !0 = \{!1\} \]
\[ !1 = \{!2\} \]
\[ !2 = \{\} \]

create (unresolved) node for \( !1 \)
Constructing a graph, top-down

!0 = {!1}
!1 = {!2}
!2 = {}
Constructing a graph, top-down

!0 = !{!1}
!1 = !{!2}
!2 = !{ }

create node for !2
Constructing a graph, top-down

\[ !0 = !\{!1\} \]
\[ !1 = !\{!2\} \]
\[ !2 = !\{\} \]

replace temporary node for !2 with real node, resolving !1 and !0
Constructing a graph, top-down

\[ 0 = \{!1\} \]
\[ 1 = \{!2\} \]
\[ 2 = \{\} \]

that was a lot of RAUW and malloc traffic...
Constructing a graph, bottom-up

0 = {1}
1 = {2}
2 = {}
Constructing a graph, bottom-up

\[ \begin{align*}
!0 &= \{!1\} \\
!1 &= \{!2\} \\
!2 &= \{} \\
\end{align*} \]

create node for !2
Constructing a graph, bottom-up

\[ !0 = \{!1\} \]
\[ !1 = \{!2\} \]
\[ !2 = \{\} \]

create node for \(!1\)
Constructing a graph, bottom-up

\[ \begin{align*}
!0 &= \{!1\} \\
!1 &= \{!2\} \\
!2 &= \{\} \\
\end{align*} \]

create node for \( !0 \)
Constructing a graph, bottom-up

0 = \{!1\}
!1 = \{!2\}
!2 = \{}
Constructing a cycle of uniqued nodes

\[
\begin{align*}
!0 &= !\{!1\} \\
!1 &= !\{!2\} \\
!2 &= !\{!0\}
\end{align*}
\]

building a cycle of uniqued nodes \textbf{requires} temporary nodes
Not every node should be uniqued

- graphs intentionally defeat uniquing when they want distinct nodes
- `!alias.scopes` need distinct root nodes
- `DILexicalBlocks` lack naturally discriminating operands
- cycles of uniqued nodes need forward references and RAUW
- cycles of uniqued nodes "look" distinct
- we don't solve graph isomorphism
**distinct nodes are more efficient**

- **distinct** nodes are not uniqued
  
  \[!1 = \text{distinct} \!\{\} \]
  
  \[!2 = \text{distinct} \!\{\} \]

- note: self-references are automatically **distinct**
  
  \[!1 = !\{!1\} \quad \rightarrow \quad !1 = \text{distinct} \!\{!1\} \]

- no re-uniquing penalty when operands change

- never require use-lists (or RAUW support)
Constructing cyclic graphs efficiently

!0 = distinct !{!1}
!1 = !{!2}
!2 = !{!0}

we can do better with distinct nodes
Constructing cyclic graphs efficiently

!0 = distinct {!1}
!1 = {!2}
!2 = {!0}

create node for !0, with a dangling operand
Constructing cyclic graphs efficiently

\[!0 = \text{distinct } !{!1}\]
\[!1 = !{!2}\]
\[!2 = !{!0}\]

create node for \(!2\)
Constructing cyclic graphs efficiently

\[ \begin{align*}
!0 &= \text{distinct } !\{!1\} \\
!1 &= !\{!2\} \\
!2 &= !\{!0\}
\end{align*} \]
Constructing cyclic graphs efficiently

\[ !0 = \text{distinct } !\{!1\} \]
\[ !1 = !\{!2\} \]
\[ !2 = !\{!0\} \]

patch operand(s) for \( !0 \)
Constructing cyclic graphs efficiently

0 = distinct 1
1 = 2
2 = 0

- careful scheduling avoids malloc traffic and RAUW
- partial support in lib/Linker; not done in BitcodeReader (yet)
Grab bag: other major LTO optimizations

- Metadata lazy-loaded (in bulk); new LTO API to expose it
- avoided lib/Linker quadratic memory leak into LLVMContext from globals with appending linkage
- debug info requires fewer MCSymbols (and they're cheaper)
- Value has dropped a couple of pointers
What progress have we made?

runtime and peak memory usage of ld, when linking executables from 3.6 (r240577) source tree

<table>
<thead>
<tr>
<th>compiler version</th>
<th>small (verify-uselistorder)</th>
<th>medium (llvm-lto)</th>
<th>large (clang)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5 (r232544)</td>
<td>48s</td>
<td>10m 35s</td>
<td>25m 41s</td>
</tr>
<tr>
<td></td>
<td>2.27GB</td>
<td>22.8GB</td>
<td>75.6GB</td>
</tr>
<tr>
<td>3.6 (r240577)</td>
<td>38s</td>
<td>8m 32s</td>
<td>19m 45s</td>
</tr>
<tr>
<td></td>
<td>1.40GB</td>
<td>15.1GB</td>
<td>35.9GB</td>
</tr>
<tr>
<td>3.7 (r247539)</td>
<td>35s</td>
<td>7m 52s</td>
<td>18m 10s</td>
</tr>
<tr>
<td></td>
<td>0.79GB</td>
<td>9.15GB</td>
<td>19.3GB</td>
</tr>
<tr>
<td>ToT (r250621)</td>
<td>34s</td>
<td>7m 37s</td>
<td>16m 23s</td>
</tr>
<tr>
<td></td>
<td>0.73GB</td>
<td>8.11GB</td>
<td>17.2GB</td>
</tr>
<tr>
<td>3.5 vs. ToT</td>
<td>1.4x</td>
<td>1.4x</td>
<td>1.6x</td>
</tr>
<tr>
<td></td>
<td>3.1x</td>
<td>2.8x</td>
<td>4.4x</td>
</tr>
</tbody>
</table>

self-hosted clang/libLTO, using ld64-253.2 from Xcode 7 on a 2013 Mac Pro with 32GB RAM
What's left in LLVM?

- use more **distinct** nodes; take more advantage of them
- richer syntax for scoped debug info nodes
- fine-grained lazy-loading of debug info metadata
  - debug info graphs need to be sliceable (link only what's used)
- MC-layer diet v2 (I'm looking at **you**, MCRelaxableFragment)
- leave debug info types out of LTO!
Debug Information

• provides a mapping from **source code** → **binary program**

• stored in extra sections in the `.o` files

```
StringRef.o

.text:
  _ZN4llvm9StringRef:

.debug_info:
  class StringRef {
    ...

.debug_line:
  0x10 StringRef.cpp 23
  0x20 StringRef.h 128
  ...
```
Where does it go?

Option 1: linker leaves debug info in the .o files
Where does it go?

**Option 1:** linker leaves debug info in the .o files
- fast linking — slow debugging

Which file has the definition of `StringRef`?
Where does it go?

**Option 2:** linker links debug info together with the executable

- typically done on Linux
- very long link times
Where does it go?

**Option 2:** linker links debug info together with the executable

- typically done on Linux
- very long link times
- split DWARF
  - relocatable **skeleton** linked with executable
  - bulk in external `.dwo`

```text
bin/clang

.debug_info:
    class StringRef {
    ...

.debug_line:
    x+0x0 StringRef.cpp 23
    x+0x10 StringRef.h 128
    ...
```
Where does it go?

**Option 3:** debug info archived separately from executable

StringRef.o

- `.text:`
  
  \_ZN4llvm9StringRef:

- `.debug_info:`
  
  class StringRef {
  ...

- `.debug_line:`
  
  0x10 StringRef.cpp 23
  0x20 StringRef.h 128
  ...

PassManager.o

- `.text:`

- `.debug_info:`

- `.debug_line:`

Cc1_main.o

- `.text:`

- `.debug_info:`

- `.debug_line:`
Where does it go?

**Option 3:** debug info archived separately from executable

1. dsymutil (Darwin)
2. dwp (Linux)
Why is `clang.dSYM` 1.2GB?

- the problem is type information, specifically, **redundant type information**:
  - `#include "llvm/ADT/StringRef.h"` at `-g` recursively pulls in ~46KB of types into each `.o` file and there are ~1500 `.o` files
(llvm-)dsymutil

• a new linker for debug information built on top of LLVM

• dsymutil collects debug info from all the .o files and generates a single .dSYM bundle with all the debug info and accelerator tables for fast lookup

• dsymutil performs ODR type uniquing for C++
(llvm-)dsymutil

<table>
<thead>
<tr>
<th></th>
<th>clang.dSYM</th>
<th>clang.dSYM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-no-odr</td>
<td></td>
</tr>
<tr>
<td>Regular</td>
<td>1.2G</td>
<td>413M</td>
</tr>
<tr>
<td>LTO</td>
<td>369M</td>
<td>388M</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
-flimit-debug-info
(also known as -fno-standalone-debug)

• emit C++ class types only in the .o file that has the vtable of the class or an explicit template instantiation and **forward declarations** everywhere else

• only C++ classes with vtables / explicit template instantiations

• every .o file and (3rd-party) library must be built with debug info

• debugger must scan every .o file for the definition of `StringRef`
  (LLDB does not even support that)

• Darwin and FreeBSD default to **-fstandalone-debug**
<table>
<thead>
<tr>
<th></th>
<th>_build/lib</th>
<th>clang.dSYM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ninja clang</strong> (1561 targets)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone</td>
<td>4.1G</td>
<td>413M</td>
</tr>
<tr>
<td>Limited</td>
<td>3.1G</td>
<td>402M</td>
</tr>
<tr>
<td><strong>LTO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone</td>
<td>5.1G</td>
<td>388M</td>
</tr>
<tr>
<td>Limited</td>
<td>3.9G</td>
<td>387M</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
Clang Modules

- Clang Modules are a saner alternative to textual `#include`
- think of them as **precompiled headers** + additional semantics
- on disk: `.pcm` file with the **serialized Clang AST** of header files
  - Darwin: built implicitly and stored in a global module cache
  - Linux: typically built explicitly
Module Debugging

- build **Debug Info** together with the **Clang Module**
- new driver option: `-gmodules`
  
  ```
  cc1: -dwarf-ext-refs -fmodule-format=obj
  ```

  - emit COFF/ELF/Mach-O Module containers with a **.clang_ast** section holding the AST.
  - emit full debug information for every type in the module
  - debug info contributes ~15% of the **.pcm** size

Module debugging also works with precompiled headers
Reminder: -flimit-debug-info

use: forward declaration

TableGen.o
.text:
call _ZN4llvm9StringRef...

.debug_info:
namespace {
  class StringRef;
}

definition

StringRef.o
.text:
...

.debug_info:
namespace llvm {
  class StringRef {
    StringRef(const char*);
  }
  ...

Module Debugging

use: forward declaration

TableGen.o

.text:
call _ZN4llvm9StringRef...

.debug_info:
module LLVM_Utils {
    module ADT {
        namespace {
            class StringRef;
        }
    }
}
dwo_name = LLVM_Utils.pcm
dwo_id = <module_hash>

LLVM_Utils.pcm

.clang_ast:
...

.debug_info:
module LLVM_Utils {
    module ADT {
        namespace llvm {
            class StringRef {
                StringRef(const char*);
            }
        }
    }
}

metadata for rebuilding module for header file

split DWARF for locating module debug info on disk
dsymutil and Clang Modules

- dsymutil clones the debug info from all imported modules into the .dSYM bundle bottom-up
- meanwhile using “ODR” type uniquing to resolve all forward declarations
- top-level modules are unique: this works for C, C++ and Objective-C
- consumers of the resulting .dSYM need not know about modules

```cpp
class clang.dSYM {
    .debug_info:
    module Darwin {
        module C {
            module stdint {
                ...
            }
            ...
        }
    }
    module std {
        ...
        module vector {
            ...
        }
    }
    module LLVM_Utils {
        ...
    }
    StringRef(const char*)
}
```
dsymutil and Clang Modules

<table>
<thead>
<tr>
<th>ninja clang (1561 targets)</th>
<th>Wall Clock</th>
<th>_build/lib</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standalone</strong></td>
<td>7m 30s</td>
<td>4.1G</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
## dsymutil and Clang Modules

<table>
<thead>
<tr>
<th>Table Title</th>
<th>Wall Clock</th>
<th>_build/lib</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standalone</strong></td>
<td>7m 30s</td>
<td>4.1G</td>
</tr>
<tr>
<td><strong>LTO</strong></td>
<td>26m 39s</td>
<td>5.1G</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
# dsymutil and Clang Modules

<table>
<thead>
<tr>
<th>ninja clang (1561 targets)</th>
<th>Wall Clock</th>
<th>_build/lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>7m 30s</td>
<td>4.1G</td>
</tr>
<tr>
<td>Limited</td>
<td>7m 23s</td>
<td>3.1G</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LTO</th>
<th>Wall Clock</th>
<th>_build/lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>26m 39s</td>
<td>5.1G</td>
</tr>
<tr>
<td>Limited</td>
<td>26m 05s</td>
<td>3.9G</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
## dsymutil and Clang Modules

<table>
<thead>
<tr>
<th>ninja clang (1561 targets)</th>
<th>Wall Clock</th>
<th>_build/lib</th>
<th>modules-cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>7m 30s</td>
<td>4.1G</td>
<td></td>
</tr>
<tr>
<td>Limited</td>
<td>7m 23s</td>
<td>3.1G</td>
<td></td>
</tr>
<tr>
<td>-fmodules</td>
<td>6m 28s</td>
<td>7.2G</td>
<td>322M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LTO</th>
<th>Wall Clock</th>
<th>_build/lib</th>
<th>modules-cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>26m 39s</td>
<td>5.1G</td>
<td></td>
</tr>
<tr>
<td>Limited</td>
<td>26m 05s</td>
<td>3.9G</td>
<td></td>
</tr>
<tr>
<td>-fmodules</td>
<td>31m 41s</td>
<td>8.9G</td>
<td>322M</td>
</tr>
</tbody>
</table>

Measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
# dsymutil and Clang Modules

<table>
<thead>
<tr>
<th></th>
<th>Wall Clock</th>
<th>_build/lib</th>
<th>modules-cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>7m 30s</td>
<td>4.1G</td>
<td></td>
</tr>
<tr>
<td>Limited</td>
<td>7m 23s</td>
<td>3.1G</td>
<td></td>
</tr>
<tr>
<td><code>-fmodules</code></td>
<td>6m 28s</td>
<td>7.2G</td>
<td>322M</td>
</tr>
<tr>
<td><code>-gmodules</code></td>
<td>4m 54s</td>
<td>1.2G</td>
<td>368M</td>
</tr>
<tr>
<td><strong>LTO</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone</td>
<td>26m 39s</td>
<td>5.1G</td>
<td></td>
</tr>
<tr>
<td>Limited</td>
<td>26m 05s</td>
<td>3.9G</td>
<td></td>
</tr>
<tr>
<td><code>-fmodules</code></td>
<td>31m 41s</td>
<td>8.9G</td>
<td>322M</td>
</tr>
<tr>
<td><code>-gmodules</code></td>
<td>20m 35s</td>
<td>1.6G</td>
<td>369M</td>
</tr>
</tbody>
</table>

Measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM, clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link.
dsymutil and Clang Modules

<table>
<thead>
<tr>
<th></th>
<th>Wall Clock</th>
<th>_build/lib</th>
<th>modules-cache</th>
<th>clang.dSYM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standalone</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ninja clang</td>
<td>7m 30s</td>
<td>4.1G</td>
<td></td>
<td>413M</td>
</tr>
<tr>
<td>(1561 targets)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limited</td>
<td>7m 23s</td>
<td>3.1G</td>
<td></td>
<td>402M</td>
</tr>
<tr>
<td>-fmodules</td>
<td>6m 28s</td>
<td>7.2G</td>
<td>322M</td>
<td>564M</td>
</tr>
<tr>
<td>-gmodules</td>
<td>4m 54s</td>
<td>1.2G</td>
<td>368M</td>
<td>453M</td>
</tr>
<tr>
<td>LTO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone</td>
<td>26m 39s</td>
<td>5.1G</td>
<td></td>
<td>388M</td>
</tr>
<tr>
<td>Limited</td>
<td>26m 05s</td>
<td>3.9G</td>
<td></td>
<td>387M</td>
</tr>
<tr>
<td>-fmodules</td>
<td>31m 41s</td>
<td>8.9G</td>
<td>322M</td>
<td>381M</td>
</tr>
<tr>
<td>-gmodules</td>
<td>20m 35s</td>
<td>1.6G</td>
<td>369M</td>
<td>407M</td>
</tr>
</tbody>
</table>

measured on a 2013 Mac Pro with 12 cores at 2.7GHz and 32GB RAM
clang r250459, X86/ARM/AArch64, RelWithDebInfo+Assertions, 1 parallel LTO link
What if consumers know about Modules?

- **LLDB** is built on top of Clang

- when evaluating an expression, LLDB
  1. loads type info from DWARF
  2. builds a Clang AST
  3. compiles and executes the Clang AST
What if consumers know about Modules?

- **LLDB** is built on top of Clang

- when evaluating an expression, LLDB
  1. loads type info from DWARF
  2. builds a Clang AST
  3. compiles and executes the Clang AST

- **a module-aware LLDB**
  - imports the type’s AST from the Clang Module
Questions?