How to update debug info
in compiler transformations

Adrian Prantl
Vedant Kumar
1. What is debug info?
2. Managing source locations
3. Tooling for writing debug info tests
1. What is debug info?
2. Managing source locations
3. Tooling for writing debug info tests
Users of debug info

Debuggers

Profilers

Coverage

Instrumentation
LLVM Debug Info crash course
Kinds of debug information

Source Locations

Inlining Information

Data Types

Source Variables and their Locations

```cpp
call void @llvm.dbg.declare(metadata i32* %X,
   metadata !11,
   !DIExpression())

!11 = !DILocalVariable(name: "X", scope: !4,
   file: !1, line: 2,
   type: !12)
```
Kinds of debug information

Source Locations

```assembly
text
load i32*, i32** %x.addr, !dbg !14
: !14 = !DILocation(line: 22, column: 4, scope: !0)
```

Inlining Information

```assembly
text
!23 = !DILocation(line: 2, column: 8, scope: !24, inlinedAt: !25)
```

Source Variables and their Locations

```assembly
text
call void @llvm.dbg.declare(metadata i32* %X, metadata !11, !DIExpression())
: !11 = !DILocalVariable(name: "X", scope: !4, file: !1, line: 2, type: !12)
```

Data Types

```assembly
text
!1 = !DIBasicType(name: "int", size: 32, align: 32, encoding: DW_ATE_signed))
```
Kinds of debug information

Source Locations

load i32*, i32** %x.addr, !dbg !14

!14 = !DILocation(line: 22, column: 4, scope: !0)

Inlining Information

!23 = !DILocation(line: 2, column: 8, scope: !24, inlinedAt: !25)

Source Variables and their Locations

call void @llvm.dbg.declare(metadata i32* %X, metadata !11, !DIExpression())

!11 = !DILocalVariable(name: "X", scope: !4, file: !1, line: 2, type: !12)

Data Types

!1 = !DIBasicType(name: "int", size: 32, align: 32, encoding: DW_ATE_signed)

Don't worry about this
Kinds of debug information

**Source Locations**

```
load i32*, i32** %x.addr, !dbg !14

!14 = !DILocation(line: 22, column: 4, scope: !0)
```

**Source Variables and their Locations**

```
call void @llvm.dbg.declare(metada i32* %X,
    metadata !11,
    !DIExpression())

!11 = !DILocalVariable(name: "X", scope: !4,
    file: !1, line: 2,
    type: !12)
```

**Inlining Information**

```
!23 = !DILocation(line: 2, column: 8, scope: !24,
    inlinedAt: !25)
```

**Data Types**

```
!1 = !DIBasicType(name: "int", size: 32,
    align: 32, encoding: DW_ATE_signed))
```

**Don't worry about this**
Kinds of debug information

Source Locations

```
load i32*, i32** %x.addr, !dbg !14
:
!14 = !DILocation(line: 22, column: 4, scope: !0)
```

Inlining Information

```
!23 = !DILocation(line: 2, column: 8, scope: !24, inlinedAt: !25)
```

Source Variables and their Locations

```
call void @llvm.dbg.declare(metadata i32* %X, metadata !11, !DIExpression())
:
!11 = !DILocalVariable(name: "X", scope: !4, file: !1, line: 2, type: !12)
```

Data Types

```
!1 = !DIBasicType(name: "int", size: 32, align: 32, encoding: DW_ATE_signed)
```

Stay tuned!
1. What is debug info?
2. Managing source locations
3. Tooling for writing debug info tests
Source Locations

- Debug info maps instructions to source locations
- An instruction `DebugLoc` contains file, line/column, scope and inline information
- Represented as `DILocation` LLVM metadata

```assembly
load i32*, i32** %x.addr, !dbg !14
: !14 = !DILocation(line: 22, column: 4, scope: !0)
```
Compiler's job is to delete, reorder, merge, sink/hoist, clone, & create instructions to maximize performance.

How to keep a meaningful mapping to the source code?

- **Spoiler alert.** It's not generally possible to unambiguously map source location to optimized code.
- Different consumers have different priorities.
- Treat debug info preservation as an optimization problem.
Principles for updating debug info
Principles for updating debug info

1. Make no misleading statements about the program
   - An optimized version of a program should appear to take the same conditions as the unoptimized version (assuming full determinism)
   - Don't speculate! No info is better than info that is only correct sometimes.
Principles for updating debug info

1. Make no misleading statements about the program
   - An optimized version of a program should appear to take the same conditions as the unoptimized version (assuming full determinism)
   - Don't speculate! No info is better than info that is only correct sometimes.

2. Provide as much information as possible
   - When it's not misleading to preserve a source location, do so!
What *can* the compiler do?

Our menu of options

- 🤗 Keep the original location
- 🐐 Merge
- 🗑 Delete
What *can* the compiler do?

Our menu of options

😊 Keep the original location

🔍 Merge

🗑 Delete
What *can* the compiler do?

Our menu of options

🤗 Keep the original location

⚗ Merge

🗑 Delete

Scopes correspond to nested {} in C++ and determine which variables are visible.

!DILocation(line: 22, column: 4, scope: !25)
!DILocation(line: 25, column: 8, scope: !25)
∩
!DILocation(line: 0, column: 0, scope: !25)

Lines start counting at 1. Line 0 denotes «no source location».
What can the compiler do?

Our menu of options

😊 Keep the original location

🐑 Merge

🗑️ Delete

%foo = add i32 %i, i32 1, !dbg !15
Block-local transformations
Replace or Expand

Examples taken from DAGCombine and Legalizer.
Replace or Expand
Try to keep the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete
Replace or Expand

Try to keep the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!

Would keeping create misleading information?

\[
\text{Replace or Expand}
\]

Try to keep the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!

Would keeping create misleading information?

\[
\begin{align*}
\text{(add } x x) & \rightarrow (\text{shl } x 1) \\
\text{(ctpop } x_{0..127}) & \rightarrow (\text{add } (\text{ctpop } x_{0..63}) (\text{ctpop } x_{64..127}))
\end{align*}
\]
Replace or Expand
Try to keep the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!

Does not change conditions which appear taken. Preserve!
Instruction reordering

Example taken from the MI instruction scheduler.
Instruction reordering
Try to keep the debug location.

```
%sum = add i32 %x, %y
%prod = mul i32 %x, %x
```

Possible Actions
1. Keep
2. Merge
3. Delete
Instruction reordering
Try to keep the debug location.

Would keeping create misleading information?

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don’t mislead!
2. Preserve!
Instruction reordering
Try to keep the debug location.

1. Don't mislead!
2. Preserve!

Possible Actions
1. Keep
2. Merge
3. Delete

%sum = add i32 %x, %y
⋮
%prod = mul i32 %x, %x

Does not change conditions which appear taken. Preserve!
Inter-block transformations
Fold block into unique predecessor

Example taken from SimplifyCFG.
Fold block into unique predecessor
Try to keep the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don’t mislead!
2. Preserve!

```
A:
  !dbg !1
  store i32 1, i32* %x
  ...
```

```
:   !dbg !1
  store i32 1, i32* %x
  ...
```
Fold block into unique predecessor
Try to keep the debug location.

Does not change conditions which appear taken. Preserve!

Principles
1. Don't mislead!
2. Preserve!

Possible Actions
1. Keep
2. Merge
3. Delete
Merging loads/stores

Example taken from MergedLoadStoreMotion.
Merging loads/stores
Try to keep the debug locations.

Possible Actions
1. Keep
2. Merge
3. Delete
Merging loads/stores
Try to keep the debug locations.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!

Can't do it yet.
Debug info consumers need to pick one location.
Merging loads/stores
Try to merge the debug locations.

Use `Instruction::applyMergedLocation()`.
Speculative execution

start:
  br i1 %cond, label %A, label %B

A:
  %and = and i1 %cond, %mask
  br label %B

B:
  %phi = phi i1 [ %and, %A ], [ %cond, %start ]

Example taken from SimplifyCFG.
Speculative execution
Try to keep the debug location.

```
start:
  br i1 %cond, label %A, label %B

A:
  !dbg !1
  %and = and i1 %cond, %mask
  br label %B

B:
  %phi = phi i1 [ %and, %A ], [ %cond, %start ]
```

Must not do it.
Makes it look like %cond is always true!

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!
Speculative execution
Try to merge the debug location.

A:
!dbg !1
%and = and i1 %cond, %mask
br label %B

B:
%phi = phi i1 [ %and, %A ], [ %cond, %start ]

Can't do it.
Nothing to merge the location with.

Possible Actions
1. Keep
2. Merge
3. Delete
Speculative execution
Try to merge the debug location.

Possible Actions
1. Keep
2. Merge
3. Delete

Principles
1. Don't mislead!
2. Preserve!

Try to merge the debug location.

start:
    br i1 %cond, label %A, label %B

A:
    !dbg !1
    %and = and i1 %cond, %mask
    br label %B

B:
    %phi = phi i1 [ %and, %A ], [ %cond, %start ]

Drop the location.

Use Instruction::dropLocation().
1. What is debug info?
2. Managing source locations
3. Tooling for writing debug info tests
Requirements for a debug info test

- A debug info test validates source locations after a transformation
- Requires reduced IR to exercise the correct transformation
- Requires reduced debug info metadata (possibly initially generated by a frontend)
Converting tests into debug info tests

- Easier than ever to test IR or MIR transformations with debug info present
- Use `opt -debugify` to attach debug info metadata to IR instructions
- Use `llc -run-pass=mir-debugify` to do the same to MIR instructions
- MIR debugify can also be applied during GlobalISel
- Documentation
  - [https://llvm.org/docs/HowToUpdateDebugInfo.html](https://llvm.org/docs/HowToUpdateDebugInfo.html)
define void @f(i32* %x) {
    store i32 1, i32* %x
    ret void
}
After `opt -debugify -debugify-level=locations`

```assembly
define void @f(i32* %x) !dbg !7 {
  store i32 1, i32* %x, !dbg !8
  ret void, !dbg !9
}

!7 = !DISubprogram(name: "f", ...)
!8 = !DILocation(line: 1, ...)
!9 = !DILocation(line: 2, ...)
```
Writing a *good* debug info test

- Check that the *correct* location is used, not just *any* location
- Do not hardcode metadata numbers into **CHECK** lines (they change!)
- Minimize the amount of metadata present (debugify helps with this)
- Try `opt -strip -debugify` to pare down to synthetic locations only
define void @f(i32* %x) !dbg !7 {
    ; CHECK: store i32 1, i32* %x, !dbg !8
    store i32 1, i32* %x, !dbg !8
    ret void, !dbg !9
}
define void @f(i32* %x) !dbg !7 {
  ; CHECK: store i32 1, i32* %x, !dbg ![[storeLoc:[0-9]+]]
  store i32 1, i32* %x, !dbg !8
  ret void, !dbg !9
}

; CHECK: ![[storeLoc]] = !DILocation(line: 1
Recap
• Debug info has a large and diverse set of applications
• Every transformation can affect the source location mapping
• Simple guidelines available to help manage source locations
• Tools available to help write clean IR or MIR-based debug info tests