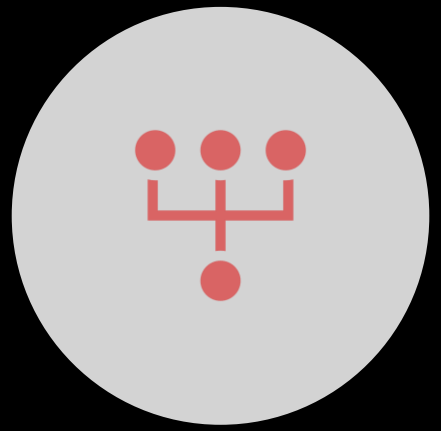
# Heterogeneous Debug Metadata

Scott Linder
Software Development Engineer at AMD

# Three Challenges

MULTIPLE LOCATIONS



DEPENDENCIES AND COMPOSITES

CONTEXT

# A Step Back…

- LLVM debug metadata describes **where source variables are**
- Metadata nodes identify source objects

```
!0 = !DILocalVariable(name: "var")
```

- Intrinsics act as “anchors” in the instruction stream:

```
void @llvm.dbg.addr(<SSA>, <Var>, <Expr>)
void @llvm.dbg.value(<SSA>, <Var>, <Expr>)
```

# Current Example

```
void func(void) {
    int var;
    // …
}
```

```
define void @func() {
  %var.alloca = alloca i32
  call void @llvm.dbg.addr(metadata ptr %var.alloca,
                           metadata !0,
                           metadata !DIExpression())

  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```

# Example: With “kill”

```
void func(void) {
    int var;
    // …
}
```

Variable Live Over This Range

```
define void @func() {
  %var.alloca = alloca i32
  call void @llvm.dbg.addr(metadata ptr %var.alloca,
                           metadata !0,
                           metadata !DIExpression())
  ; …
  call void @llvm.dbg.value(metadata ptr undef,
                            metadata !0,
                            metadata !DIExpression())
  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```

# Multiple Locations

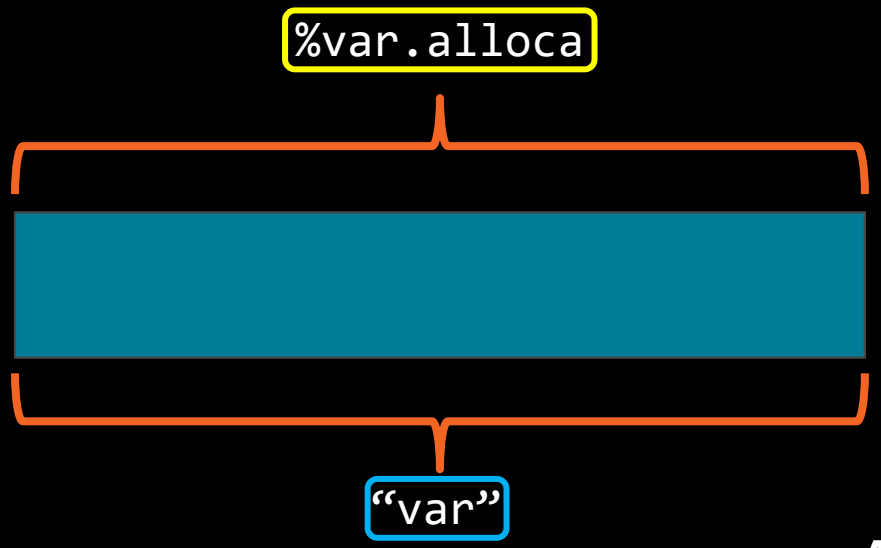
- Optimizing compiler may allocate parallel storage for value
- Example:
  - Value stored in memory
  - Read into register during loop body
  - Compiler proves loop does not write to the value
  - Elides store to memory after loop body
- Debug agent must write to both locations
  - DIExpression-based metadata explicitly forbids describing this

# Dependencies and Composites

- LLVM builds composite values using “fragments” (`DW_OP_LLVM_fragment`)
- Fragment mechanism incomplete
  - Each fragment describes where it fits into a larger whole
  - Cannot operate on the complete whole

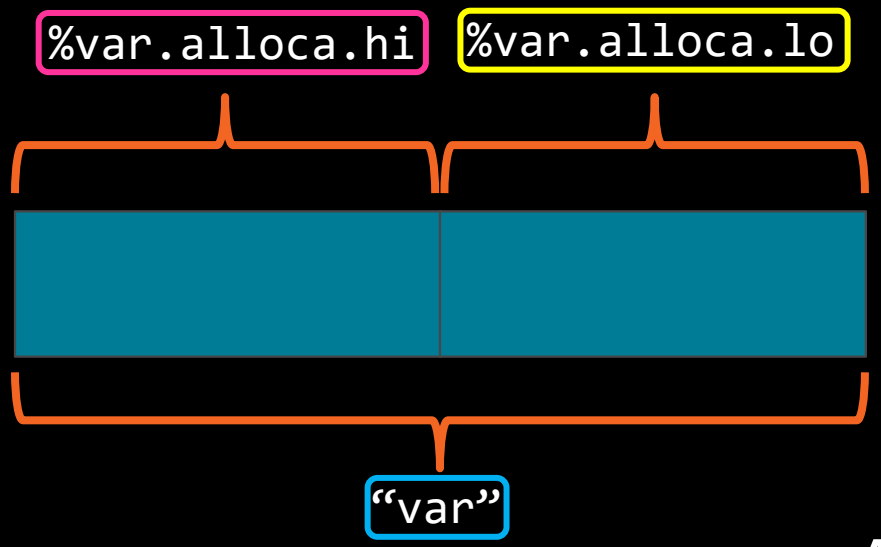
# Simple Example

```
define void @func() {
  %var.alloca = alloca i32
  call void @llvm.dbg.addr(metadata ptr %var.alloca,
                           metadata !0,
                           metadata !DIExpression())

  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```



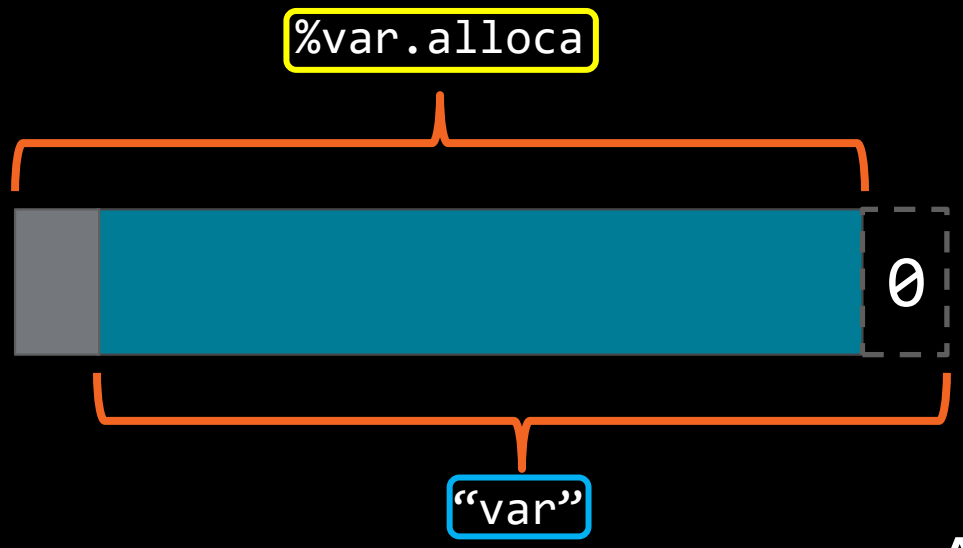
# Optimized Into Fragments

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.addr(metadata ptr %var.alloca.lo,
                           metadata !0,
                           metadata !DIExpression(DW_OP_LLVM_fragment, 0, 16))
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.addr(metadata ptr %var.alloca.hi,
                           metadata !0,
                           metadata !DIExpression(DW_OP_LLVM_fragment, 16, 16))

  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```



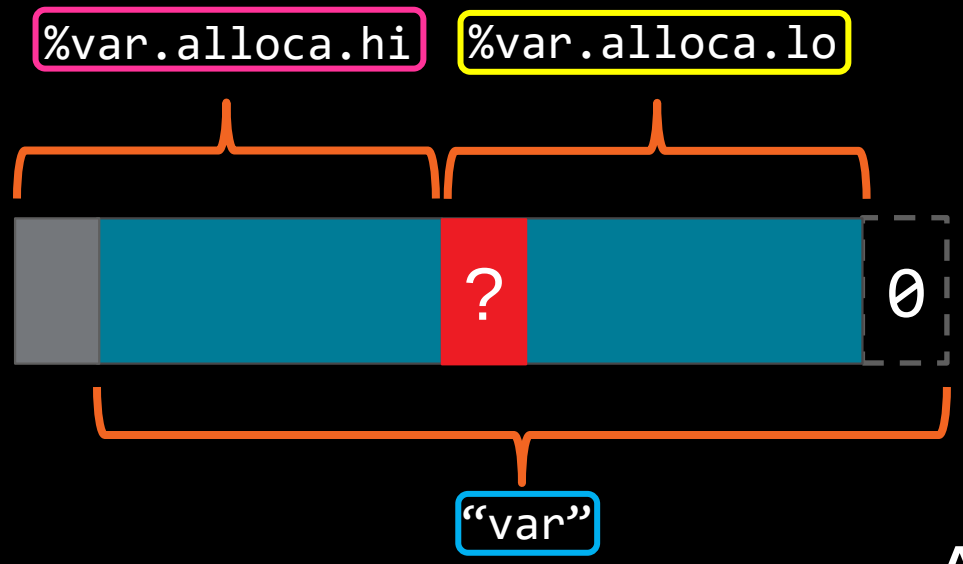
# Induction Variable Optimization

```
define void @func() {
  %var.alloca = alloca i32
  call void @llvm.dbg.value(
    metadata ptr %var.alloca,
    metadata !0,
    metadata !DIExpression(DW_OP_constu, 1, DW_OP_shl, DW_OP_stack_value))
  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```



# Fragmenting an Induction Variable

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.value(metadata ptr %var.alloca.lo, metadata !0,
    metadata !DIExpression(DW_OP_constu, 1, DW_OP_shl, DW_OP_LLVM_fragment, 0, 16))
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.value(metadata ptr %var.alloca.hi, metadata !0,
    metadata !DIExpression(???, DW_OP_LLVM_fragment, 16, 16))

  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
```



# Dependencies and Composites (cont.)

- DIExpression unfactorable
  - Can only depend on LLVM Value-like entities
  - No symbolic dependencies

# A Case for Factoring Expressions

```
define void @func() {
  %var.alloca.lo = alloca i16
  %var.alloca.hi = alloca i16
  ; …
  ; …
  loop.outer:
    ; …
    ; …
    loop.inner:
      ; …
      br %loop.inner
    ; …
    br %loop.outer
  ; …
  ret void
}
```

# Only a Single Location

```
define void @func() {
  %var.alloca.lo = alloca i16
  %var.alloca.hi = alloca i16
  ; …
  ; …
  loop.outer:
    ; …
    ; …
    loop.inner:
      ; …
      br %loop.inner
    ; …
    br %loop.outer
  ; …
  ret void
}
```

Composite(%var.alloca.lo, %var.alloca.hi)

# Single Dependency in Multiple Locations

```
define void @func() {
  %var.alloca.lo = alloca i16
  %var.alloca.hi = alloca i16
  ; …
  %var.lo = load i16, ptr %var.alloca.lo
  loop.outer:
    ; …
    ; …
    loop.inner:
      ; …
      br %loop.inner
    ; …
    br %loop.outer
  ; …
  ret void
}
```

Composite(%var.alloca.lo, %var.alloca.hi)

Composite(%var.alloca.lo, %var.alloca.hi)
Composite(%var.lo, %var.alloca.hi)

Composite(%var.alloca.lo, %var.alloca.hi)

# Multiple Dependencies in Multiple Locations

```
define void @func() {
  %var.alloca.lo = alloca i16
  %var.alloca.hi = alloca i16
  ; …
  %var.lo = load i16, ptr %var.alloca.lo
  loop.outer:
    ; …
    %var.hi = load i16, ptr %var.alloca.hi
    loop.inner:
      ; …
      br %loop.inner
    ; …
    br %loop.outer
  ; …
  ret void
}
```

Composite(%var.alloca.lo, %var.alloca.hi)

Composite(%var.alloca.lo, %var.alloca.hi)
Composite(%var.lo, %var.alloca.hi)

Composite(%var.alloca.lo, %var.alloca.hi)
Composite(%var.lo, %var.alloca.hi)
Composite(%var.alloca.lo, %var.hi)
Composite(%var.lo, %var.hi)

Composite(%var.alloca.lo, %var.alloca.hi)
Composite(%var.lo, %var.alloca.hi)

Composite(%var.alloca.lo, %var.alloca.hi)

# Context

- Meaning of a DIExpression depends on context
- Pseudo-operations like DW_OP_stack_value affect interpretation of all inputs
- There is a proposal to add yet more flags to handle more cases
  - Further complicates compiler’s task
- Flags interact poorly with multiple inputs, dependencies and composites

# Challenges for Making Incremental Changes

- Changes ripple through several intrinsics
- Internals are exposed, requiring changes at many call-sites
- DIArgList begins to address dependencies/composites
  - Currently limited to only “stack_value” expressions
  - Lifting restrictions requires yet more stateful flags
  - Still doesn’t address factoring

# A New Approach

- Supports symbolic dependencies
  - DIFragment captures non-source symbolic entities
  - DIObject acts as abstract base for DIVariable and DIFragment
- Supports multiple locations
  - DILifetime captures many-to-one relationship of locations and objects
- Expressions for variables are always complete, never built from many fragments
- Facilitates simple, mechanical updates which do not incur cost of inspecting/updating context

# Supports Explicit Lifetime of Variables

```
define void @func() {
  %var.alloca = alloca i32
  call void @llvm.dbg.def(metadata !1, metadata ptr %var.alloca)
  ; …
  call void @llvm.dbg.kill(metadata !1)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0,
  location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i32)))
```

# Supports Multiple Locations

```
define void @func() {
  call void @llvm.dbg.def(metadata !1, …)
  ; …
  call void @llvm.dbg.def(metadata !2, …)
  ; …
  call void @llvm.dbg.kill(metadata !1)
  ; …
  call void @llvm.dbg.kill(metadata !2)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0, location: !DIExpr(…))
!2 = distinct !DILifetime(object: !0, location: !DIExpr(…))
```

# Optimized Into Fragments

```
define void @func() {
  %var.alloca.lo = alloca i16

  %var.alloca.hi = alloca i16

  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0, location: !DIExpr(???))
```

# Optimization Reflected with DIFragment Objects

```
define void @func() {
  %var.alloca.lo = alloca i16

  %var.alloca.hi = alloca i16

  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0, location: !DIExpr(???))



!2 = distinct !DIFragment()


!4 = distinct !DIFragment()
```

# Each DIFragment Described Independently

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.def(metadata !3, metadata ptr %var.alloca.lo)
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.def(metadata !5, metadata ptr %var.alloca.hi)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0, location: !DIExpr(???))



!2 = distinct !DIFragment()
!3 = distinct !DILifetime(object: !2,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16))
!4 = distinct !DIFragment()
!5 = distinct !DILifetime(object: !4,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16)))
```

# Original Expression Mechanically Updated with Dependencies

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.def(metadata !3, metadata ptr %var.alloca.lo)
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.def(metadata !5, metadata ptr %var.alloca.hi)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0,
  location: !DIExpr(DIOpArg(1, i16), DIOpArg(0, i16), DIOpComposite(2, i32)
                    DIOpAddr(0), DIOpDeref(i32)),
  argObjects: {!2, !4})
!2 = distinct !DIFragment()
!3 = distinct !DILifetime(object: !2,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16))
!4 = distinct !DIFragment()
!5 = distinct !DILifetime(object: !4,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16)))
```

# Original Expression Mechanically Updated with Dependencies

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.def(metadata !3, metadata ptr %var.alloca.lo)
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.def(metadata !5, metadata ptr %var.alloca.hi)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0,
  location: !DIExpr(DIOpArg(1, i16), DIOpArg(0, i16), DIOpComposite(2, i32)),
  argObjects: {!2, !4})
!2 = distinct !DIFragment()
!3 = distinct !DILifetime(object: !2,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16))
!4 = distinct !DIFragment()
!5 = distinct !DILifetime(object: !4,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16)))
```

# Mechanical Updates Compose Naturally

```
define void @func() {
  %var.alloca = alloca i32, align 4
  call void @llvm.dbg.def(metadata !1, metadata ptr %var.alloca)
  ; …
  call void @llvm.dbg.kill(metadata !1)
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0,
  location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i32), DIOpShl(1)))
```

# Optimizer Can Ignore DIOpShl

```
define void @func() {
  %var.alloca.lo = alloca i16
  call void @llvm.dbg.def(metadata !3, metadata ptr %var.alloca.lo)
  %var.alloca.hi = alloca i16
  call void @llvm.dbg.def(metadata !5, metadata ptr %var.alloca.hi)
  ; …
  ret void
}

!0 = !DILocalVariable(name: "var")
!1 = distinct !DILifetime(object: !0,
  location: !DIExpr(DIOpArg(1, i16), DIOpArg(0, i16), DIOpComposite(2, i32), DIOpShl(1)),
  argObjects: {!2, !4})
!2 = distinct !DIFragment()
!3 = distinct !DILifetime(object: !2,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16))
!4 = distinct !DIFragment()
!5 = distinct !DILifetime(object: !4,
                          location: !DIExpr(DIOpReferrer(ptr), DIOpDeref(i16)))
```

# Heterogeneous Compute

- Address spaces require even more flags to control implicit indirections
- Dependencies/composites arise more routinely with very large vector registers which are used both for SIMD and SIMT purposes

# DISCLAIMER AND ATTRIBUTIONS

AMD