What would it take to remove debug intrinsics?

Jeremy Morse

Variable locations require llvm::Value’s and a position

```
define dso_local i32 @_Z3fooiiib(i32 %a, i32 %b, i32 %c, i1 zeroext %d) local_unnamed_addr #0 !dbg !9 {
entry:
  call void @llvm.dbg.value(metadata i32 %a, metadata !15, metadata !DIExpression()), !dbg !21
  call void @llvm.dbg.value(metadata i32 %b, metadata !16, metadata !DIExpression()), !dbg !21
  call void @llvm.dbg.value(metadata i32 %c, metadata !17, metadata !DIExpression()), !dbg !21
  call void @llvm.dbg.value(metadata i1 %d, metadata !18, metadata !DIExpression(...)), !dbg !21
  %add = add nsw i32 %b, %a, !dbg !22
  call void @llvm.dbg.value(metadata i32 %add, metadata !19, metadata !DIExpression()), !dbg !21
  %mul = mul nsw i32 %add, %c, !dbg !23
  call void @llvm.dbg.value(metadata i32 %mul, metadata !20, metadata !DIExpression()), !dbg !21
  %add1 = add nsw i32 %mul, 10
  %spec.select = select i1 %d, i32 %add1, i32 %mul, !dbg !24
  call void @llvm.dbg.value(metadata i32 %spec.select, metadata !20, metadata !DIExpression()), !dbg !21
  ret i32 %spec.select, !dbg !25
}
```

Why is this bad?

- In-band signalling mixes data and metadata -- generated code can change if you give –g on the command line.
  - –Block size changes depending on presence of debug-info
  - –Peephole optimisations
- Poor performance
  - –Up to 50% opt time, 30% of a large LTO link

A new variable-location design:

- Objectives:
  - Compile-time efficient
  - No interference with optimisations
  - Identical output to current design
- We have an initial prototype design (see our discourse post)
  - Changes to LLVM's instruction API are required

https://discourse.llvm.org/t/rfc-instruction-api-changes-needed-to-eliminate-debug-intrinsics-from-ir/68939

The instruction API as a language

- Sometimes debug-info as instructions is a useful abstraction
- Sometimes it isn't

```
join_blocks a b

insert_instr_at_start

foreach_instr_in_block
  if_property_present
    move_somewhere
```

```
BB->getInstrList().splice(OtherBlock, BB.begin(), BB.end());

FooInst->insertBefore(OtherBlock.begin());

for (auto &Instr : BB) {
  if (SomePredicateFunc(Instr)) {
    Instr->moveBefore(OtherBlock, OtherBlockIt);
  }
}
```

The moveBefore problem

- If we move %mul into %bb1, should the debug-info travel with it?
- If the multiply is being hoisted, then no, we're just moving a computation
- If the two blocks are being merged, then yes, debug-info should travel
- Knowing which requires information about the intention from the caller

```
bb1:
  %add = add nsw i32 %b, %a, !dbg !22
  br label %bb2

bb2:
  call void @llvm.dbg.value(metadata, i32 0, metadata (...))
  %mul = mul nsw i32 %add, %c, !dbg !23
```

The head insertion problem

- If we sink %add into bb2, should it come before or after the debug-info?
- If we’re sinking because %add is redundant, it doesn’t matter
- If %add immediately precedes bb2, it should come before the debug-info
- Knowing which requires information about the intention from the caller

```
bb1:
  %add = add nsw i32 %b, %a, !dbg !22
  br i1 %cond, label %retblock, label %bb2

bb2:
  call void @llvm.dbg.value(metadata i32 0, metadata (...))
  %mul = mul nsw i32 %add, %c, !dbg !23
  ...
```

Abstraction: does this transform preserve execution order?

```
bb1:
  %foo = add i32 %0, %1
  br label %bb2

bb2:
  %bar = sub i32 %foo, %2
  br label %bb3
```

$\longrightarrow$

```
bb1:
  %foo = add i32 %0, %1
  %bar = sub i32 %foo, %2
  br label %bb3
```

```
bb1:
  %foo = add i32 %0, %1
  br i1 %cond, label %bb2, label %bb3

bb2:
  %bar = sub i32 %foo, %2
  br label %bb4

bb3:
  %baz = sub i32 %foo, %3
  br label %bb4
```

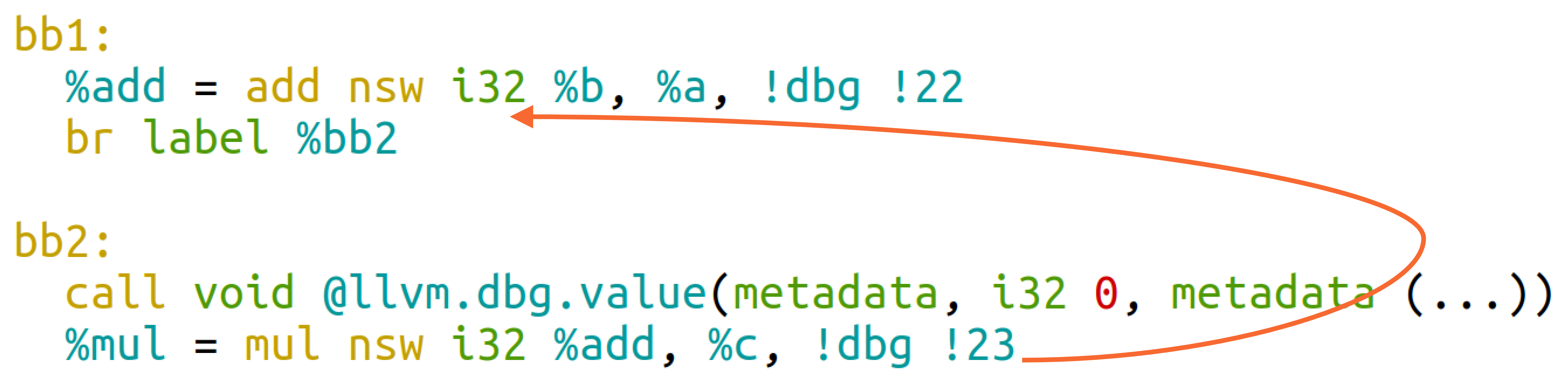
$\longrightarrow$

```
bb2:
  %foo.1 = add i32 %0, %1
  %bar = sub i32 %foo.1, %2
  br label %bb4

bb3:
  %foo.2 = add i32 %0, %1
  %baz = sub i32 %foo.2, %3
  br label %bb4
```

Proposal one: intentionality of moves

- moveBeforeBreaking: move instruction while breaking sequence
- moveBeforePreserving: move instruction while preserving sequence



```
bb1:
  %add = add nsw i32 %b, %a, !dbg !22
  br label %bb2

bb2:
  call void @llvm.dbg.value(metadata, i32 0, metadata (...))
  %mul = mul nsw i32 %add, %c, !dbg !23
```

Proposal two: stuff bits into iterators

```
BasicBlock::iterator It =
  BB->getFirstInsertionPt();
SomeInstruction->insertBefore(It);
```

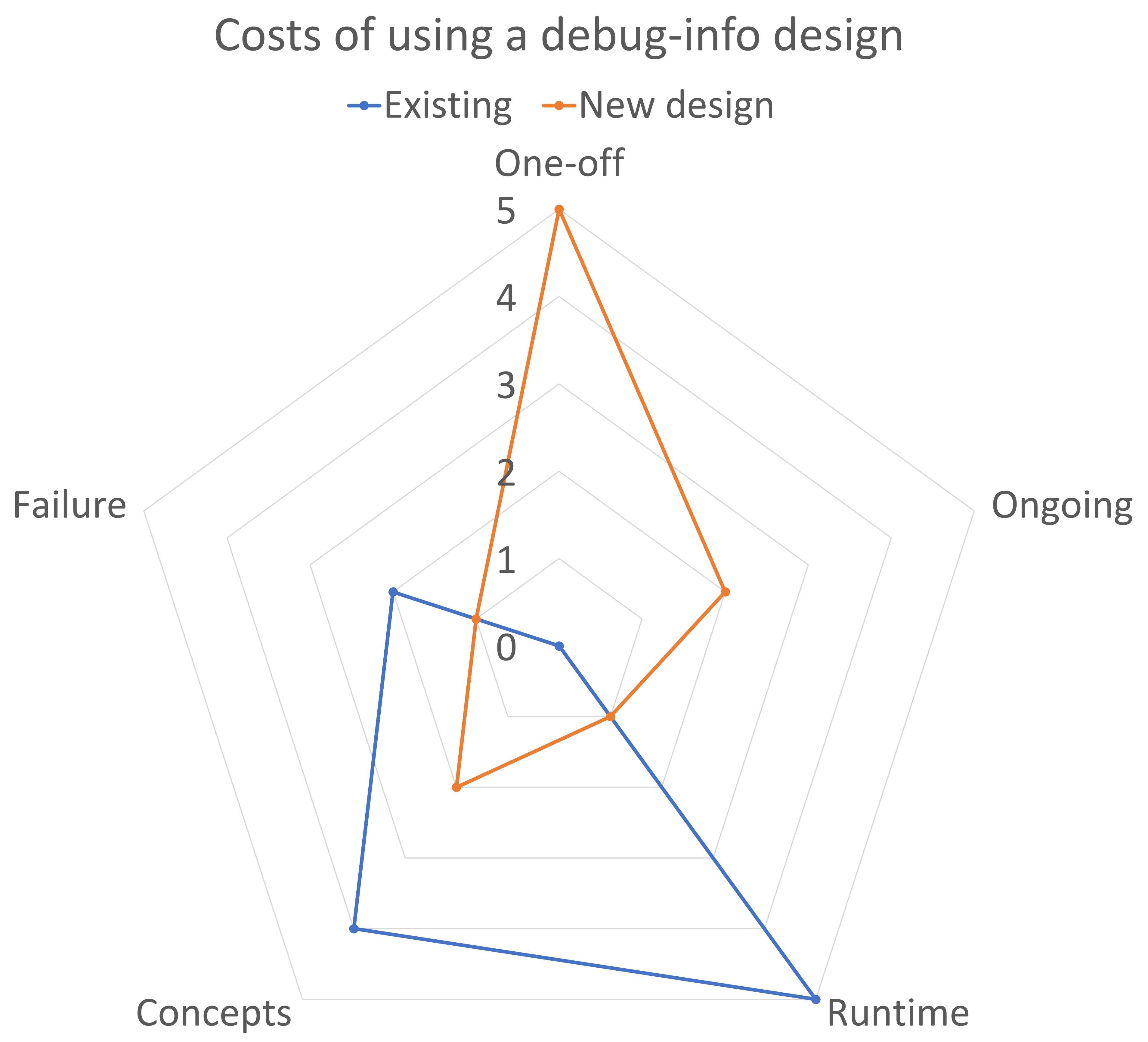
```
bb1:
  %add = add nsw i32 %b, %a, !dbg !22
  br i1 %cond, label %retblock, label %bb2

bb2:
  call void @llvm.dbg.value(metadata i32 0, metadata (...))
  %mul = mul nsw i32 %add, %c, !dbg !23
  ...
```

```
BasicBlock::iterator It;
for (auto &Inst : *BB) {
  if (FilterMatchesInst(Inst)) {
    It = Inst->getIterator();
    break;
  }
}
SomeInstruction->moveBefore(It);
```

Many places we can put the costs

- One-off costs
- Ongoing development costs
- Runtime costs
- Concepts costs
- Failure costs



Summary

- We can save up to 30% of compile-time in debug-info LTO builds
- Information about the intention of a transformation is needed
- Knowing whether the execution sequence of instructions is preserved is sufficient
- There are a few ways to implement this in LLVM
- (I reckon my proposal is the most balanced!)

Thank you!