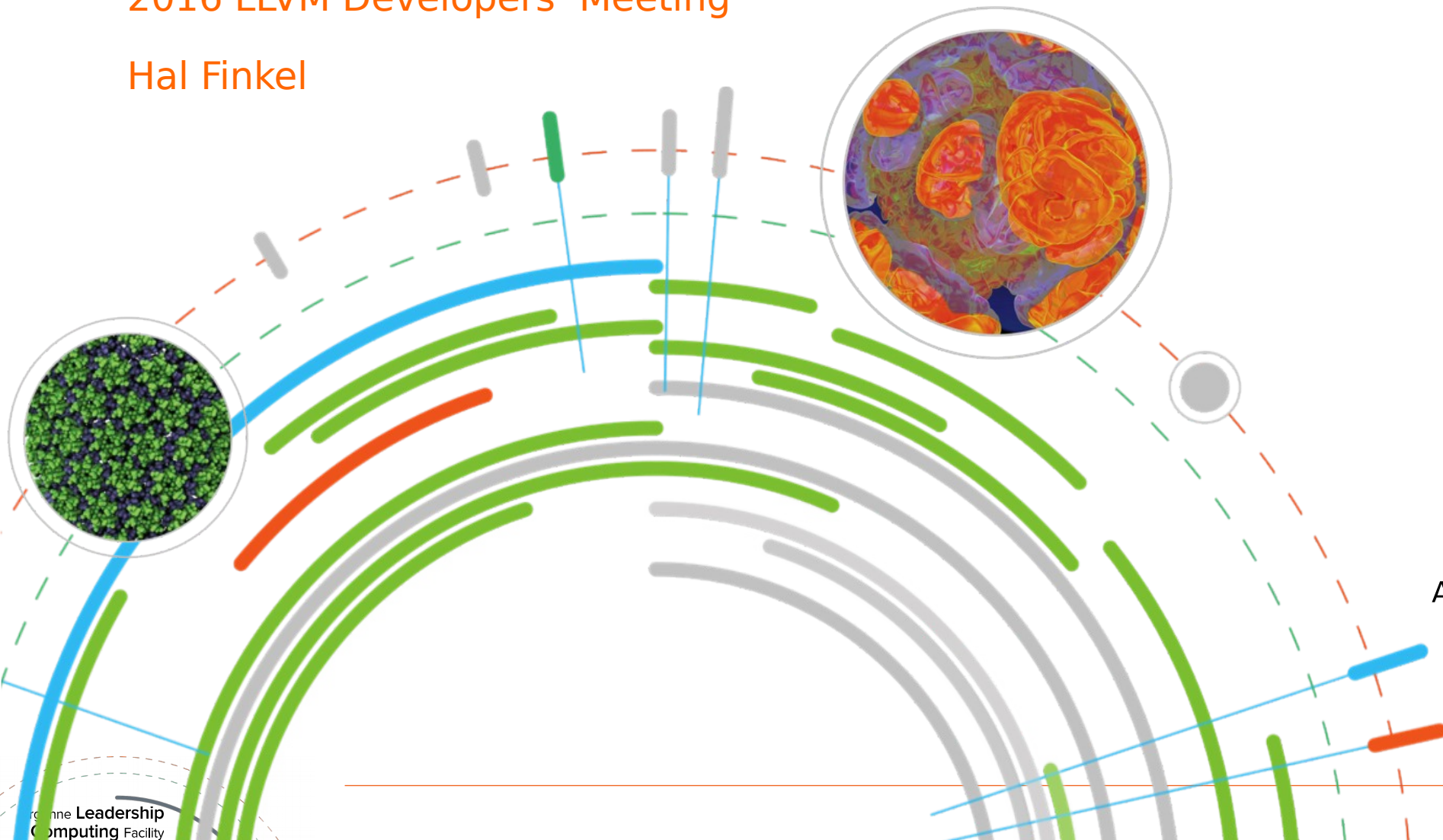


Intrinsics, Metadata, and Attributes: The story continues!

2016 LLVM Developers' Meeting

Hal Finkel



Argonne **Leadership**
Computing Facility



Goals of This Presentation:

- ✓ To review LLVM's concepts of intrinsics, metadata and attributes
- ✓ To explain how our metadata representation has changed
- ✓ To introduce some recent addition to these families
- ✓ To discuss how they should, and should not, be used
- ✓ To explain how Clang uses these new features
- ✓ To discuss how these capabilities might be expanded in the future

5/27/2014

The LLVM Compiler Infrastructure Project

The LLVM Compiler Infrastructure

Site Map:

[Overview](#)
[Features](#)
[Documentation](#)
[Command Guide](#)
[FAQ](#)
[Publications](#)
[LLVM Projects](#)
[Open Projects](#)
[LLVM Users](#)
[Bug Database](#)
[LLVM Logo](#)
[Blog](#)
[Meetings](#)

Download!

Download now:
[LLVM 3.4](#)
[All Releases](#)
[APT Packages](#)
[Win Installer](#)

View the open-source
[license](#)

Search this Site

Search!

Useful Links

Mailing Lists:
[LLVM-announce](#)
[LLVM-dev](#)
[LLVM-bugs](#)
[LLVM-commits](#)
[LLVM-branch-commits](#)
[LLVM-testresults](#)

IRC Channel:
[irc.oftc.net/#llvm](#)

Dev. Resources:

<http://llvm.org>

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be [used to build them](#). The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a [research project](#) at the [University of Illinois](#), with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of [commercial and open source](#) projects as well as being widely used in [academic research](#). Code in the LLVM project is licensed under the ["UIUC" BSD-Style license](#).

The primary sub-projects of LLVM are:

1. The **LLVM Core** libraries provide a modern source- and target-independent [optimizer](#), along with [code generation support](#) for many popular CPUs (as well as some less common ones). These libraries are built around a [well specified](#) code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are [well documented](#), and it is particularly easy to invent your own language (or port an existing compiler) to use [LLVM as an optimizer and code generator](#).
2. **Clang** is an "LLVM native" C/C++/Objective-C compiler, which

Latest LLVM Release!

Jan 6, 2014: LLVM 3.4 is now [available for download!](#) LLVM is publicly available under an open source [License](#). Also, you might want to check out [the new features](#) in SVN that will appear in the next LLVM release. If you want them early, [download LLVM](#) through anonymous SVN.

ACM Software System Award!

LLVM has been awarded the **2012 ACM Software System Award!** This award is given by ACM to one software system worldwide every year. LLVM is [in highly distinguished company!](#) Click on any of the individual recipients' names on that page for the detailed citation describing the award.

Upcoming Releases

Onward to 3.5!

Developer Meetings

Proceedings from past meetings

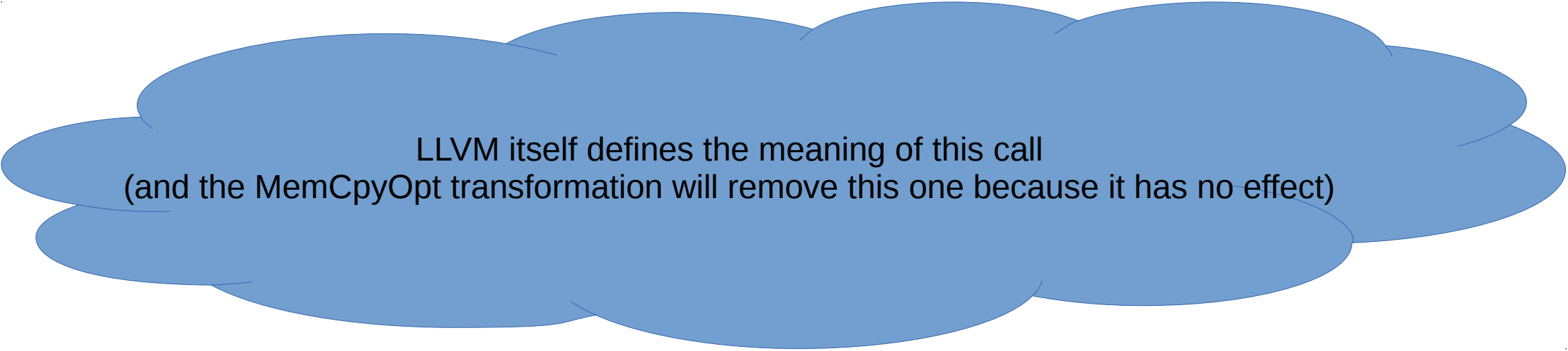
- [April 7-8, 2014](#)
- [Nov 6-7, 2013](#)
- [April 29-30, 2013](#)
- [November 7-8, 2012](#)
- [April 12, 2012](#)
- [November 18, 2011](#)
- [September 2011](#)

1/4

Background: Intrinsic

Intrinsics are “internal” functions with semantics defined directly by LLVM. LLVM has both target-independent and target-specific intrinsics.

```
define void @test6(i8 *%P) {  
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %P, i8* %P, i64 8, i32 4, i1 false)  
  ret void  
}
```

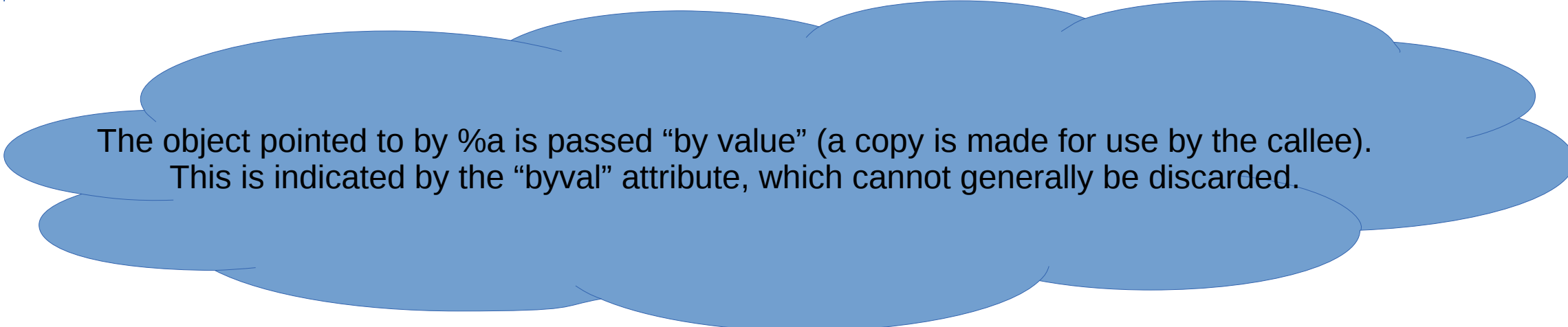


LLVM itself defines the meaning of this call
(and the MemCpyOpt transformation will remove this one because it has no effect)

Background: Attributes

Properties of functions, function parameters and function return values that are part of the function definition and/or callsite itself.

```
define i32 @foo(%struct.x* byval %a) nounwind {  
  ret i32 undef  
}
```

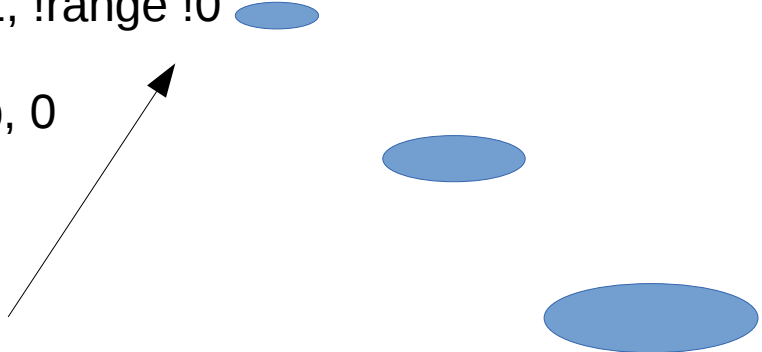


The object pointed to by %a is passed “by value” (a copy is made for use by the callee). This is indicated by the “byval” attribute, which cannot generally be discarded.

Background: Metadata

Metadata represents optional information about an instruction (or module) that can be discarded without affecting correctness.

```
define zeroext i1 @_Z3fooPb(i8* nocapture %x) {  
entry:  
  %a = load i8* %x, align 1, !range !0  
  %b = and i8 %a, 1  
  %tobool = icmp ne i8 %b, 0  
  ret i1 %tobool  
}  
  
!0 = !{i8 0, i8 2}
```

A diagram illustrating the use of range metadata. It features three blue oval shapes arranged in a descending staircase pattern from left to right. An arrow originates from the top-right oval and points to the 'load' instruction in the code block above. Below the code, the definition of '!0' is shown as '!{i8 0, i8 2}'.

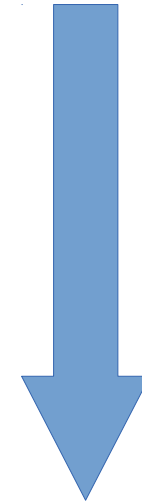
Range metadata provides the optimizer with additional information on a loaded value.
%a here is 0 or 1.

A note on expense

In what follows, we'll review these new

- **Attributes** (essentially free, use whenever you can)
- **Metadata** (comes at some cost: processing lots of metadata can slow down the optimizer)
- **Intrinsics** (intrinsics like `@llvm.assume` introduce extra instructions and value uses which, while providing potentially-valuable information, can also inhibit transformations: use judiciously!)

Cheaper



More Expensive

We now also have “operand bundles”, which are like metadata for calls, but it is illegal to drop them...
`call void @y() ["deopt"(i32 10), "unknown"(i8* null)]`

(for example, used for implementing deoptimization)

These are essentially free, like attributes, but can block certain optimizations!

Metadata Has Changed

A couple of years ago, it looked like this:

```
!0 = metadata !{ metadata !0, metadata !1 }  
!1 = metadata !{ metadata !"llvm.loop.unroll.count", i32 4 }
```



Now it looks like this:

```
!0 = distinct !{ !0, !1 }  
!1 = !{ !"llvm.loop.unroll.count", i32 4 }
```

- Metadata is now typeless in the IR (you don't need the 'metadata' keyword everywhere)
- You avoid uniquing using the 'distinct' keyword (not just by making it self-referential)
- Under the hood: Metadata nodes (!{...}) and strings (!"...") are no longer values. They have no use-lists, no type, cannot RAUW, and cannot be function-local.

For more information, see: <http://llvm.org/releases/3.6.1/docs/ReleaseNotes.html#metadata-is-not-a-value>

Metadata Has Changed

This still looks the same:

```
declare void @llvm.bar(metadata)
```

```
call void @llvm.bar(metadata !0)
```



The metadata itself is not a value
but the argument here is a value of type:
MetadataAsValue

(and the MetadataAsValue wrapper
does have a use list, etc.)

For more information, see: <http://llvm.org/releases/3.6.1/docs/ReleaseNotes.html#metadata-is-not-a-value>

Metadata On Globals

You can now put metadata on global variables:

```
@foo = external global i32, !foo !0
```

```
declare !bar !1 void @bar()
```

```
!0 = distinct !{}
```

```
!1 = distinct !{}
```



Some things that are not new any more...

Intrinsics	Metadata	Attributes
@llvm.assume	!llvm.loop.*	align
	!llvm.mem.parallel_loop_access	nonnull
	!alias.scope and !noalias	dereferenceable
	!nonnull	
	!nonnull	

Some new things...

Intrinsics	Metadata	Attributes
@llvm.masked.load.*	!dereferenceable	dereferenceable_or_null
@llvm.masked.store.*	!dereferenceable_or_null	allocsize
@llvm.masked.gather.*	!align	argmemonly
@llvm.masked.scatter.*	!unpredictable	inaccessiblememonly
		inaccessiblemem_or_argmemonly
		writeonly
		norecurse
		convergent

dereferenceable Attribute [not new any more]

Specify a known extent of dereferenceable bytes starting from the attributed pointer.

```
void foo(int * __restrict__ a, int * __restrict__ b, int &c, int n) {  
    for (int i = 0; i < n; ++i)  
        if (a[i] > 0)  
            a[i] = c*b[i];  
}
```

We can now hoist the load of the value bound to c out of this loop!

```
define void @test1(i32* noalias nocapture %a, i32* noalias nocapture readonly %b, i32* nocapture  
readonly dereferenceable(4) %c, i32 %n)
```

Clang now adds this for C++ references

And also C99 array parameters with 'static' size:

```
void test(int a[static 3]) { } produces:
```

```
define void @test(i32* dereferenceable(12) %a)
```

dereferenceable_or_null Attribute

Specify a known extent of dereferenceable bytes starting from the attributed pointer – if the pointer is known not to be null!

Not used by Clang, but covers situations like this:

```
void foo(int * __restrict__ a, int * __restrict__ b, int *c (dereferenceable_or_null), int n) {  
    if (c != nullptr) {  
        for (int i = 0; i < n; ++i)  
            if (a[i] > 0)  
                a[i] = *c*b[i];  
    }  
}
```

We can hoist the load of *c out of this loop!

```
define void @bar(i8* align 4 dereferenceable_or_null(1024) %ptr) {  
entry:  
    %ptr.gep = getelementptr i8, i8* %ptr, i32 32  
    %ptr.i32 = bitcast i8* %ptr.gep to i32*  
    %ptr_is_null = icmp eq i8* %ptr, null  
    br i1 %ptr_is_null, label %leave, label %loop  
  
...  
}
```

allocsize Attribute

Helps declare functions with some of the magic of malloc():

```
declare i8* @my_malloc(i8*, i32) allocsize(1)
```

Allocates a number of bytes given by the 2nd argument (indexing from 0, so the '1' means the 2nd argument)

```
declare i8* @my_calloc(i8*, i8*, i32, i32) allocsize(2, 3)
```

The name 'calloc' here is potentially misleading: no assumption is made about the contents of the memory (e.g. that it is zero'd)

Allocates a number of bytes given by the 3rd argument multiplied by the 4th argument.

Plans exist to use this in Clang (see review D14274), although this has not been committed yet:

```
void *my_malloc(int a) __attribute__((alloc_size(1)));  
void *my_calloc(int a, int b) __attribute__((alloc_size(1, 2)));
```

argmemonly, inaccessiblememonly, and inaccessiblemem_or_argmemonly Attributes

We've had the equivalent of argmemonly for intrinsics for a long time, but now you can get the same semantics for arbitrary functions.

```
declare i32 @func(i32 * %P) argmemonly
```

All memory accesses in the function use pointers based on its (pointer-typed) function arguments.

```
declare i32 @func() inaccessiblememonly
```

This function might access memory, but nothing that can be directly accessed from within the module.

```
declare i32 @func(i32 * %P) inaccessiblemem_or_argmemonly
```

You can guess...

The 'inaccessible memory' concept allows us to preserve ordering dependencies (i.e. side effects) while not being overly-conservative about potential pointer aliasing.

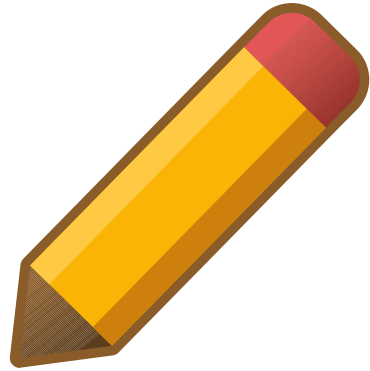
writeonly Attribute

```
declare void @a_readonly_func(i8 *) readonly  
declare void @a_writeonly_func(i8 *) writeonly
```

Balance in the Force has been restored!
(we now have both readonly and writeonly)

This function might write to memory,
but does not read from memory.

```
declare void @llvm.memset.p0i8.i32(i8* nocapture writeonly, i8, i32, i32, i1)
```



This function does not write using pointers
based on this argument
(although might write to that memory
using other aliasing pointers)

norecurse Attribute

```
define void @m() norecurse {  
    %a = call i32 @called_by_norecurse()  
    ret void  
}
```

No matter what happens in @called_by_norecurse, the program never recursively calls @m.

- In C++, main is known never to be called by user code, and so Clang marks it norecurse.
- Used to enable a few things in the optimizer (e.g. the localization of global variables, IPRA)

returned Attribute [not new, but reinvigorated]

```
declare i8* @func1(i8* returned, i32*)
```



This function always returns its first argument!

- Clang will add this attribute on some “this return” functions known to return the *this* pointer.
- This attribute is not new, but over the last year, we've taught a bunch of IR-level optimizations to understand it (and infer it).

convergent Attribute

declare void @barrier() convergent

- The problem: In many GPU SIMT models, all threads executing together (in a “warp” in NVIDIA's terminology) need to hit the **same** barrier, not just some barrier.
- Some transformations, such as loop unswitching, naturally break this requirement:

```
for (...) {  
  barrier();  
  if (cond) {  
    do_something();  
  } else {  
    do_something_else();  
  }  
}
```



```
if (cond) {  
  for (...) {  
    barrier();  
    do_something();  
  }  
} else {  
  for (...) {  
    barrier();  
    do_something_else();  
  }  
}
```

All threads used to hit the same barrier, but now they'll hit two if cond is different in different threads!

- Used by Clang when generating CUDA code – All functions/calls are conservatively marked as convergent, and then the optimizer removes the attribute when it can prove that safe.

Corresponding Metadata

Function-parameter attributes are great, but what if the value is being loaded from memory (instead of passed by value)?

```
%val = load i32*, i32** @globali32ptr, !nonnull !0  
!0 = !{}
```

These apply to loaded pointer values.

```
%val = load i32*, i32** @globali32ptr, !dereferenceable !1  
!1 = !{i64 8}
```

```
%val = load i32*, i32** @globali32ptr, !dereferenceable_or_null !2  
!2 = !{i64 16}
```

```
%val = load i32*, i32** @globali32ptr, align 8, !align !3  
!3 = !{i64 4}
```

This says something about the alignment of the loaded pointer.

This is talking about the alignment of the pointer being loaded from.

“Unpredictable” Metadata

The problem: On many modern processors, branches are better than hardware-level select/conditional-move instructions... unless the condition is unpredictable!

```
br i1 %tmp4, label %cond_true, label %UnifiedReturnBlock, !unpredictable !0
```

```
%sel = select i1 %cmp, float %div, float 2.0, !unpredictable !0
```

```
!0 = !{}
```

This is exposed in Clang via an intrinsic:

```
if (__builtin_unpredictable(x > 0))  
    foo ();
```

(kind of like `__builtin_expect`)



!llvm.loop.* Metadata [not new any more]

Fundamental question: How can you attach metadata to a loop?

LLVM has no fundamental IR construction to represent a loop, and so the metadata must be attached to some instruction; which one?

```
br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
```

```
...
```

```
!0 = distinct !{ !0, !1 }
```

```
!1 = !{ !"llvm.loop.unroll.count", i32 4 }
```

It is important to mark your loop IDs as “distinct” metadata.

The backedge branch gets the metadata

!llvm.loop.* Metadata - Source Locations

Fundamental question: How can you identify the source location of a loop?

- The source location of a loop is needed when generating optimization remarks regarding loops.
- We used to guess based on the source location of the preheader's branch (if it exists) or the branch of the loop header.

Now we have a better way: Put the debug-info source location in the metadata loop id:

```
br i1 %exitcond, label %._crit_edge, label %.lr.ph, !llvm.loop !0
```

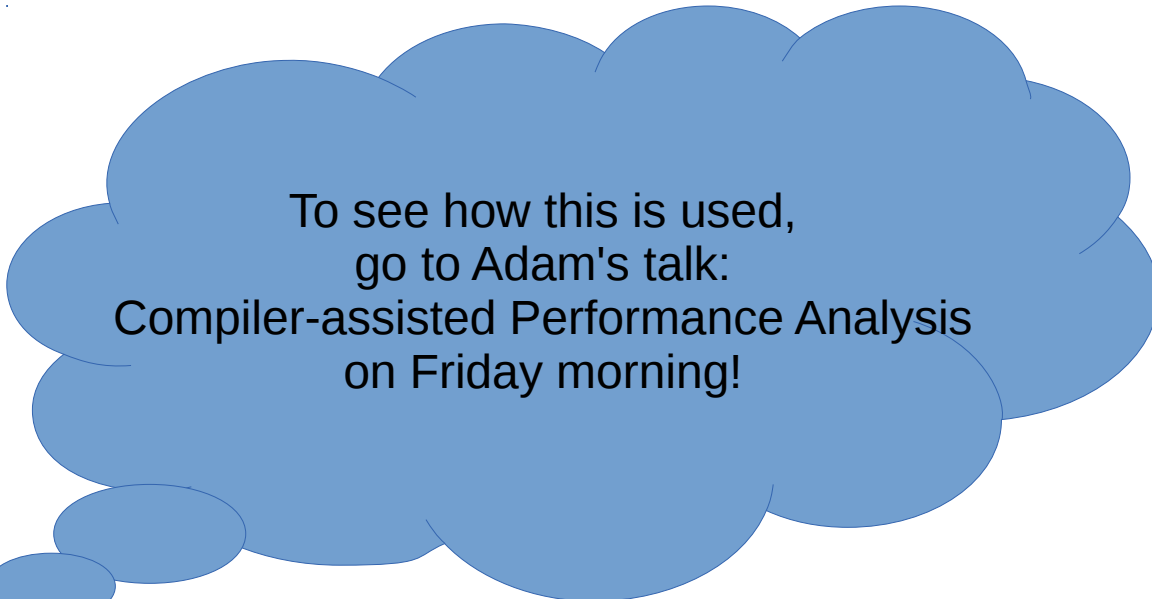
...

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

```
!1 = !{ !"llvm.loop.unroll.count", i32 4 }
```

```
!2 = !DILocation(line: 2, column: 3, scope: ...)
```

Clang now adds loop ids with source locations to all loops when optimization remarks or optimization-record saving is enabled!



To see how this is used,
go to Adam's talk:
Compiler-assisted Performance Analysis
on Friday morning!

!llvm.loop.* Metadata Optimization Control [not new any more]

- › **!llvm.loop.interleave.count**: Sets the preferred interleaving (modulo unrolling) count
- › **!llvm.loop.vectorize.enable**: Enable loop vectorization for this loop, even if vectorization is otherwise disabled
- › **!llvm.loop.vectorize.width**: Sets the preferred vector width for loop vectorization
- › **!llvm.loop.unroll.disable**: Disable loop unrolling for this loop, even when it is otherwise enabled
- › **!llvm.loop.unroll.full**: Suggest that the loop be fully unrolled (overriding the cost model)
- › **!llvm.loop.unroll.count**: Sets the preferred unrolling factor for partial and runtime unrolling (overriding the cost model)

Clang exposes these via the pragma:

```
#pragma clang loop vectorize/interleave/vectorize_width/interleave_count/unroll/unroll_count
```

!llvm.loop.* Metadata Optimization Control [some new ones]

- › **!llvm.loop.unroll.runtime.disable**: Disable runtime loop unrolling for this loop, even when unrolling is otherwise enabled
- › **!llvm.loop.distribute.enable**: Enable loop distribution (we don't have a good cost model for this yet, so it must be explicitly enabled)
- › **!llvm.loop.licm_versioning.disable**: Disables the multiversioning of this loop to enable LICM (note, however, that this pass is not yet enabled by default)

Clang exposes these via the pragma:

```
#pragma clang loop unroll/distribute
```


!invariant.group Metadata

- › The invariant.group metadata may be attached to load/store instructions.
- › Every load/store in the same group using the same pointer operand loads/stores the same value.
- › This assumption is interrupted by the `llvm.invariant.group.barrier` intrinsic.

```
store i8 42, i8* %ptr, !invariant.group !0
```

```
call void @foo(i8* %ptr)
```

```
%a = load i8, i8* %ptr, !invariant.group !0
```

```
%ptr2 = call i8* @llvm.invariant.group.barrier(i8* %ptr)
```

```
%d = load i8, i8* %ptr2, !invariant.group !0
```

%a must be 42 here!

Can't say anything here

To learn more, go to Piotr's talk:
Devirtualization in LLVM
Later this afternoon!

Intrinsics for Masked Memory Access

```
%res = call <16 x float> @llvm.masked.load.v16f32.p0v16f32 (<16 x float>* %ptr, i32 4, <16 x i1>%mask, <16 x float> %passthru)
```

The semantics are similar to this:

```
%loadlal = load <16 x float>, <16 x float>* %ptr, align 4  
%res = select <16 x i1> %mask, <16 x float> %loadlal, <16 x float> %passthru
```

Except you'll never fault for masked-off memory accesses! The loop vectorizer will generate these when hardware support is available (for example, see `isLegalMaskedLoad` in `TargetTransformInfo` – x86 AVX2/AVX-512). Important for if-conversion of conditionally-accessed arrays:

```
for (int I = ...) {  
  if (a[I] > 0)  
    ... = b[I] + 5;  
}
```

b[I] is conditionally accessed here.

Intrinsics for Masked Memory Access

But there's more:

```
declare void @llvm.masked.store.v8i32.p0v8i32 (<8 x i32> <value>, <8 x i32>* <ptr>, i32  
<alignment>, <8 x i1> <mask>)
```

```
declare <16 x float> @llvm.masked.gather.v16f32 (<16 x float*> <ptrs>, i32 <alignment>, <16 x i1>  
<mask>, <16 x float> <passthru>)
```

```
declare void @llvm.masked.scatter.v8i32 (<8 x i32> <value>, <8 x i32*> <ptrs>, i32 <alignment>,  
<8 x i1> <mask>)
```

Note: While hardware support for scatter/gather is available on some targets, the performance may still be highly variable!

Gather:

```
for (i=0; i<N; ++i)  
  x[i] = y[idx[i]];
```

Scatter:

```
for (i=0; i<N; ++i)  
  y[idx[i]] = x[i];
```

Fast-Math Flags

Here's a story about some attributes that are being replaced...

- › We used to model assumptions about floating-point numbers (e.g. `-ffast-math`) using function attributes:

attributes #0 = { ... "no-infs-fp-math"="true" "no-nans-fp-math"="true" "unsafe-fp-math"="true" }

- › But this didn't work well with LTO. So we've moved this information into flags on each instruction:

`%op1 = fadd nnan float %load1, 1.0` – Assume that the arguments and result are not NaN.

`%op1 = fadd ninf float %load1, 1.0` – Assume that the arguments and result are not +/-Inf.

`%op1 = fadd nsz float %load1, 1.0` – Assume that the sign of zero is insignificant.

`%op1 = fadd arcp float %load1, 1.0` – We can use the reciprocal rather than perform division.

`%op1 = fadd fast float %load1, 1.0` – Allow algebraically-equivalent transformations (implies all others)

Prefix Data

The problem: How to embed runtime-accessible metadata with functions (that you can get if you have the function pointer)...

```
define void @f() prefix i32 123 { ... }
```

Here's how you get the data...

```
%0 = bitcast void* () @f to i32*  
%a = getelementptr inbounds i32, i32* %0, i32 -1  
%b = load i32, i32* %a
```

Here's the data: An i32 with the value:
123

The data comes before the function address
in memory (at "index -1").

Prologue Data

The problem: How to embed arbitrary data (likely code) at the beginning of a function...

```
define void @f() prologue i8 144 { ... }
```

On x86, 144 is a nop.

This can also be used to embed metadata at the beginning of a function – to do that, you need to jump over it when the code runs...

```
%0 = type <{ i8, i8, i8* }>  
define void @f() prologue %0 <{ i8 235, i8 8, i8* @md }> { ... }
```

These first two values, on x86, are jmp .+10

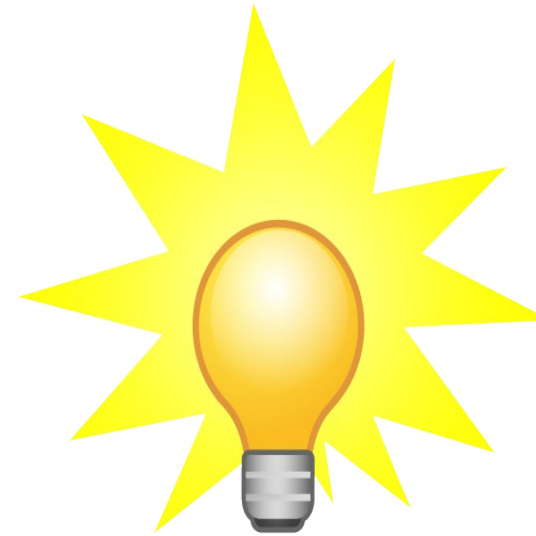
Acknowledgments

Very few of the new features covered in this talk were my work:

- ✓ Thank you to the entire LLVM community!

And for sponsoring my work:

- ✓ ALCF, ANL and DOE



→ ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

Extra Slides...

align Attribute

The align attribute itself is not new, we've had it for byval arguments, but it has now been generalized to apply to any pointer-typed argument.

```
define i32 @foo1(i32* align 32 %a) {  
entry:  
  %0 = load i32* %a, align 4  
  ret i32 %0  
}
```



This load will become align 32

Clang will emit this attribute for `__attribute__((align_value(32)))` on function arguments. When inlining, these may be transformed into `@llvm.assume`.

nonnull Attribute

A pointer-typed value is not null (on an argument or return value):

```
define i1 @nonnull_arg(i32* nonnull %i) {  
  %cmp = icmp eq i32* %i, null  
  ret i1 %cmp  
}
```

```
declare nonnull i32* @returns_nonnull_helper()  
define i1 @returns_nonnull() {  
  %call = call nonnull i32* @returns_nonnull_helper()  
  %cmp = icmp eq i32* %call, null  
  ret i1 %cmp  
}
```



These comparisons have a known result.

Clang adds this for C++ references (where the size is unknown and the address space is 0),
`__attribute__((nonnull)), __attribute__((returns_nonnull))`

Adding `__attribute__((returns_nonnull))` to LLVM's BumpPtrAllocator and MallocAllocator speeds up compilation time for bzip2.c by $(4.4 \pm 1)\%$

!llvm.mem.parallel_loop_access Metadata

What do you do when the frontend knows that certain memory accesses within a loop are independent of each other (no loop-carried dependencies), and if these are the only accesses in the loop then it can be vectorized?

```
for.body:
```

```
...
```

```
%val0 = load i32* %arrayidx, !llvm.mem.parallel_loop_access !0
```

```
...
```

```
store i32 %val0, i32* %arrayidx1, !llvm.mem.parallel_loop_access !0
```

```
...
```

```
br i1 %exitcond, label %for.end, label %for.body, !llvm.loop !0
```

```
for.end:
```

```
...
```

```
!0 = distinct !{ !0 }
```

This is a list of !llvm.loop metadata
(nested parallel loops can be expressed)

Clang exposes this via the OpenMP pragma: #pragma omp simd

!alias.scope and !noalias Metadata

An alias scope is an (id, domain), and a domain is just an id. Both !alias.scope and !noalias take a list of scopes.

; Two scope domains:

```
!0 = distinct !{ !0}
```

```
!1 = distinct !{ !1}
```

; Some scopes in these domains:

```
!2 = distinct !{ !2, !0}
```

```
!3 = distinct !{ !3, !0}
```

```
!4 = distinct !{ !4, !1}
```

; Some scope lists:

```
!5 = !{ !4} ; A list containing only scope !4
```

```
!6 = !{ !4, !3, !2}
```

```
!7 = !{ !3}
```

; These two instructions don't alias:

```
%0 = load float* %c, align 4, !alias.scope !5
```

```
store float %0, float* %arrayidx.i, align 4, !noalias !5
```

; These two instructions also don't alias (for domain !1, the set of scopes in the !alias.scope equals that in the !noalias list):

```
%2 = load float* %c, align 4, !alias.scope !5
```

```
store float %2, float* %arrayidx.i2, align 4, !noalias !6
```

; These two instructions don't alias (for domain !0, the set of scopes in the !noalias list is not a superset of, or equal to, the scopes in the !alias.scope list):

```
%2 = load float* %c, align 4, !alias.scope !6
```

```
store float %0, float* %arrayidx.i, align 4, !noalias !7
```

From restrict to !alias.scope and !noalias

An example: Preserving noalias (restrict in C) when inlining:

```
void foo(double * restrict a, double * restrict b, double *c, int i) {  
    double *x = i ? a : b;
```

```
    *c = *x;  
}
```

The actual scheme also checks for capturing (because the pointer “based on” relationship can flow through captured variables)

*x gets:
!alias.scope: 'a', 'b'
(it might be derived from 'a' or 'b')

*c gets:
!noalias: 'a', 'b'
(definitely not derived from 'a' or 'b')

*a would get:
!alias.scope: 'a'
!noalias: 'b'

The need for domains comes from making the scheme composable: When a function with noalias arguments, that has !alias.scope!/noalias metadata from an inlined callee, is itself inlined.

!nonnull Metadata

The nonnull attribute covers pointers that come from function arguments and return values, what about those that are loaded?

```
define i1 @nonnull_load(i32** %addr) {  
  %ptr = load i32** %addr, !nonnull !}  
  %cmp = icmp eq i32* %ptr, null  
  ret i1 %cmp  
}
```

The !nonnull applies to the result of the load, not the pointer operand!

The result here is known!

Will this kind of metadata be added corresponding to other function attributes? Probably.

@llvm.assume Intrinsic

Can provide the optimizer with additional control-flow-dependent truths: Powerful but use sparingly!

Why sparingly? Additional uses are added to variables you care about optimizing, and that can block optimizations. But sometimes you care more about the information being added than these optimizations: pointer alignments are a good example.

```
define i32 @foo1(i32* %a) {  
entry:  
  %0 = load i32* %a, align 4  
  
  %pprint = ptrtoint i32* %a to i64  
  %maskedptr = and i64 %pprint, 31  
  %maskcond = icmp eq i64 %maskedptr, 0  
  tail call void @llvm.assume(i1 %maskcond)  
  
  ret i32 %0  
}
```

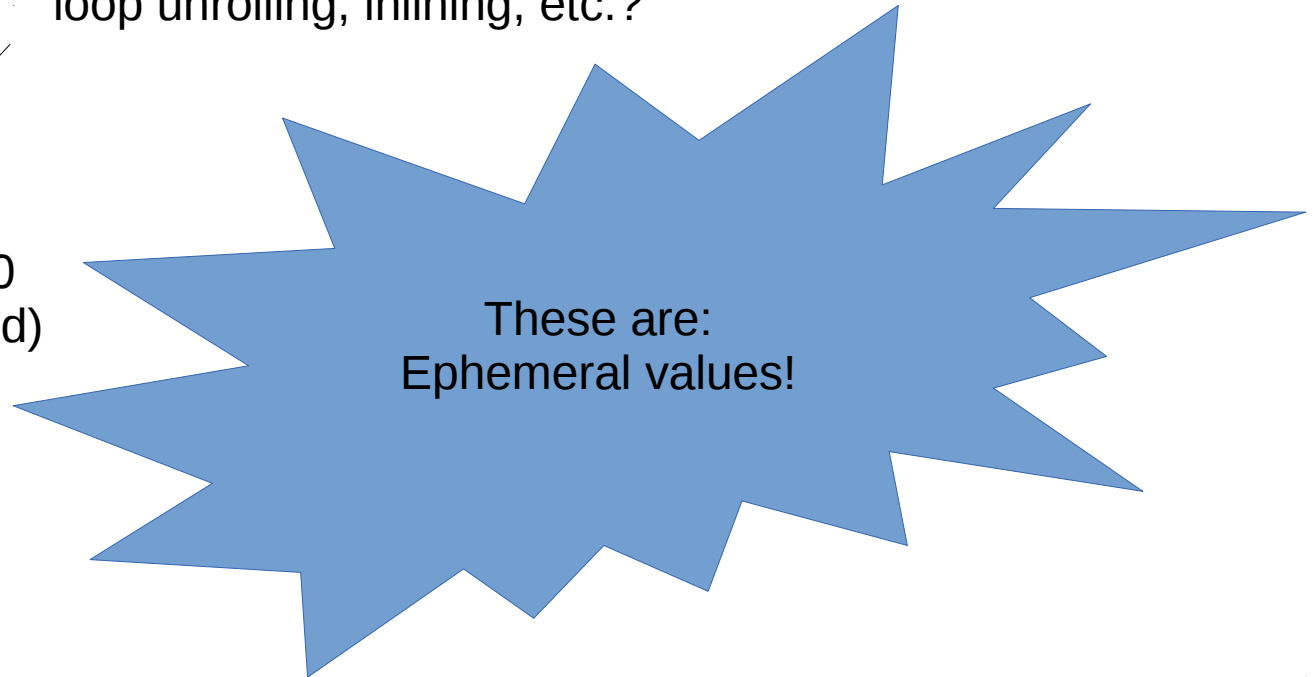
InstCombine will make this align 32, and the assume call will stay!

Assumes can be used to provide known bits (via ValueTracking used by InstCombine/InstSimplify, etc.), known ranges (via LazyValueInfo used by JumpThreading, etc.), effective loop guards (via SCEV), and more to come!

Ephemeral Values (@llvm.assume)

```
define i32 @foo1(i32* %a) {  
entry:  
  %0 = load i32* %a, align 4  
  
  %pprint = ptrtoint i32* %a to i64  
  %maskedptr = and i64 %pprint, 31  
  %maskcond = icmp eq i64 %maskedptr, 0  
  tail call void @llvm.assume(i1 %maskcond)  
  
  ret i32 %0  
}
```

But what about all of these extra values? Don't they affect loop unrolling, inlining, etc.?



Ephemeral values are collected by `collectEphemeralValues`, a utility function in `CodeMetrics`, and excluded from the cost heuristics used by the inliner, loop unroller, etc.

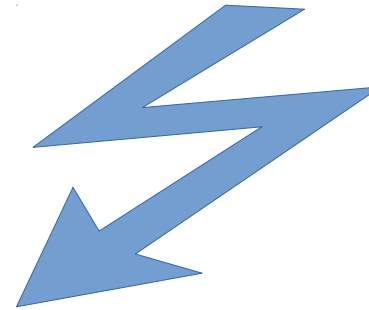
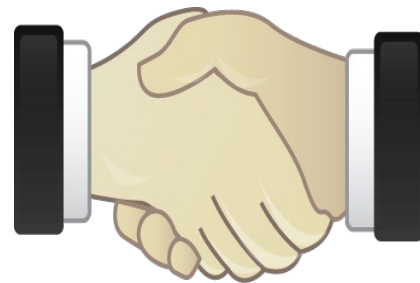
The AssumptionTracker (@llvm.assume)

Assumptions are control-flow dependent, how can you find the relevant ones?

A new module-level “invariant” analysis, the AssumptionTracker, keeps track of all of the assumes currently in the module. So finding them is easy:

```
for (auto &AssumeCall : AT->assumptions(F)) {  
  ...  
}
```

But there is a contract!



If you create a new assume, you'll need to register it with the AssumptionTracker:

```
AT->registerAssumption(CI);
```

And how can you know if an assume can be legally used to simplify a particular instruction? ValueTracking has a new utility function:

```
bool isValidAssumeForContext(const Instruction *AssumeCI, const Instruction *CxtI,  
                             const DataLayout *DL = nullptr, const DominatorTree *DT = nullptr);
```

@llvm.assume and Clang

```
int test1(int *a, int i) {  
    __builtin_assume(a != 0);
```

You can assume any boolean condition you'd like
(and we support __assume is MS-compatibility mode).

```
#ifdef _MSC_VER  
    __assume(a != 0)  
#endif  
}
```

(and '#pragma omp simd aligned(32)' too!)

```
int *m2() __attribute__((assume_aligned(64, 12)));
```

GCC-style alignment assumptions are fully supported!

```
int test3(int *a) {  
    a = __builtin_assume_aligned(a, 32);  
}
```

align attributes here

```
typedef double * __attribute__((align_value(64))) aligned_double;
```

```
void foo(aligned_double x, double * y __attribute__((align_value(32))),  
         double & z __attribute__((align_value(128)))) { };
```

```
struct ad_struct {  
    aligned_double a;
```

```
double *foo(ad_struct& x) {  
    return x.a; }
```

@llvm.assume for alignment here

A note on align_value

align_value is a GCC-style attribute, not supported by GCC, but appearing in Intel's compiler (versions 14.0+).

Why is it needed?



```
typedef double aligned_double attribute((aligned(64)));  
void foo(aligned_double *P) {  
    double x = P[0]; // This is fine.  
    double y = P[1]; // What alignment did those doubles have again?  
}
```

And this comes up a lot with loops and vectorization (on many architectures, aligned vector loads are much cheaper than potentially-unaligned ones)! Here's the semantically-correct way:

```
typedef double *aligned_double_ptr attribute((align_value(64)));  
void foo(aligned_double_ptr P) {  
    double x = P[0]; // This is fine.  
    double y = P[1]; // This is fine too.  
}
```

