Intrinsics, Metadata and Attributes: Now, more than ever!

2014 LLVM Developers' Meeting

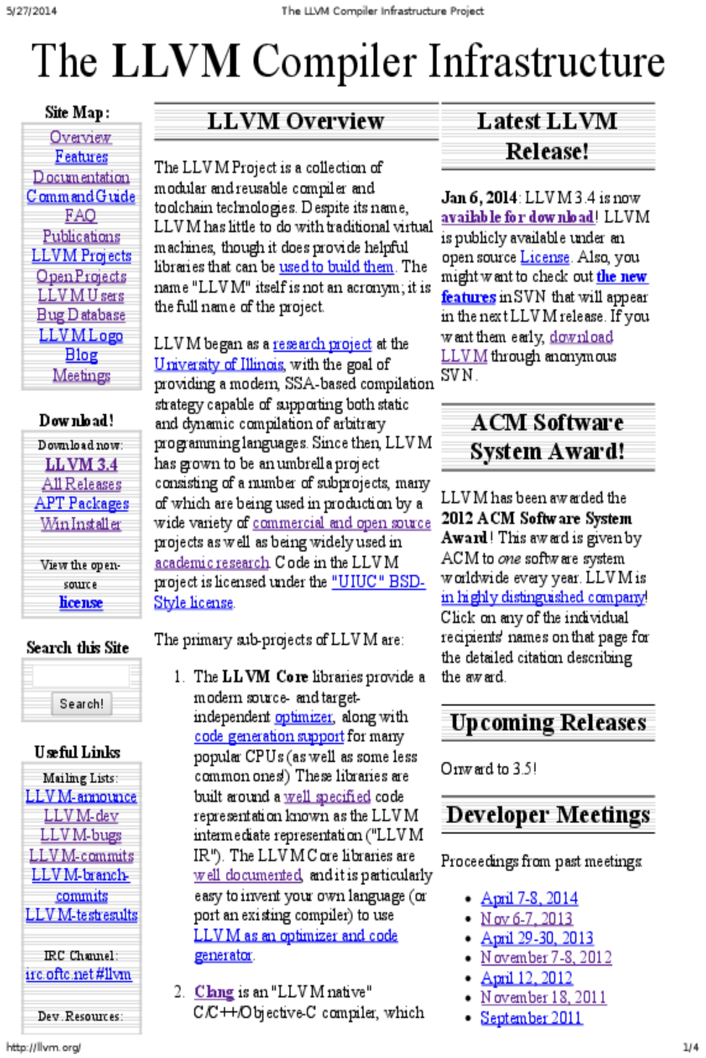
Hal Finkel

Argonne Leadership Computing Facility

Argonne NATIONAL LABORATORY

Goals of This Presentation:

- To review LLVM's concepts of intrinsics, metadata and attributes
- 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:

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

http://llvm.org/ 1/4

Background: Intrinsics

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 = metadata !{i8 0, i8 2}
```

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

Some new things...

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

Uses by Clang:

- C++ References: nonnull, dereferenceable
- __attribute__((nonnull)), __attribute__((returns_nonnull)): nonnull
- #pragma loop ... : !llvm.loop.*
- #pragma omp simd: !llvm.mem.parallel_loop_access
- __builtin_assume_aligned, __builtin_assume, __attribute__((assume_aligned)), __attribute__((align_value)), #pragma omp simd aligned: align, @llvm.assume
- Block-level __restrict__: !alias.scope and !noalias (planned)

Some new things... (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 judicially!

Cheaper

More Expensive

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 ± 1)%

dereferenceable Attribute

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)

!llvm.loop.* Metadata

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

This metadata references itself, keeping it unique!

The backedge branch gets the metadata

!llvm.loop.* Metadata (cont.)

- **!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.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 = metadata !{ metadata !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 = metadata !{metadata !0}
!1 = metadata !{metadata !1}

; Some scopes in these domains:
!2 = metadata !{metadata !2, metadata !0}
!3 = metadata !{metadata !3, metadata !0}
!4 = metadata !{metadata !4, metadata !1}

; Some scope lists:
!5 = metadata !{metadata !4} ; A list containing only scope !4
!6 = metadata !{metadata !4, metadata !3, metadata !2}
!7 = metadata !{metadata !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

  %ptrint = ptrtoint i32* %a to i64
  %maskedptr = and i64 %ptrint, 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

  %ptrint = ptrtoint i32* %a to i64
  %maskedptr = and i64 %ptrint, 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.?

These are:
Ephemeral values!

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

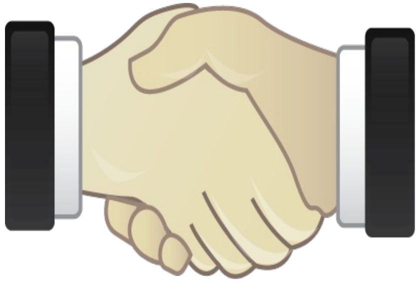
The AssumptionTracker (@llvm.assume)

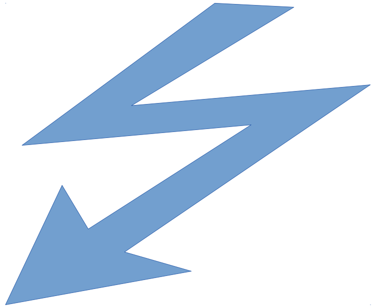
Assumptions are control-flow dependent, how can you find the relevant ones?

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);

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

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

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

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

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

GCC-style alignment assumptions are fully supported!

```
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)))) { };
```

align attributes here

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

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

@llvm.assume for alignment here

A note on align_value

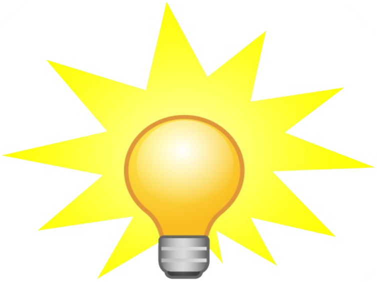
align_value is a GCC-style attribute, not supported by GCC, but appearing in Intel's compiler (versions 14.0+).

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.
}
```



Where is all of this headed?

A few things I'm pretty-sure are coming:

- A convergence of metadata and attributes (we have nonnull and !nonnull, what about !align, !dereferenceable, etc.
- Some way of tagging dereferenceable bytes with a runtime size
- Adding !noalias and !alias.scope for block-level restrict-qualified pointers in Clang (and some C++ attribute, see WG21 N4150 for some progress in this direction)

Acknowledgments

Many other people have also contributed to the features discussed in this talk, including:

- Chandler Carruth, Andy Trick, Philip Reames, Nick Lewycky, Aaron Ballman, Richard Smith, Luqman Aden, Pekka Jääskeläinen

And for sponsoring my work:

- ALCF, ANL and DOE