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
LLVM Foundation

Download!

Download now: **LLVM 5.0.0**
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

2011 LLVM Developers' Meeting

1. Talk Slides and Videos
2. Talk Abstracts
3. Poster Abstracts

- **What**: The fifth general meeting of LLVM Developers and Users.
- **Why**: To get acquainted, learn how LLVM is used, and exchange ideas.
- **When**: November 18, 2011
- **Where**: San Jose Marriott, 301 South Market Street, San Jose, CA

SPONSORED BY: QuIC, Apple, Google

We are eager to find companies to help cover travel expenses for speakers needing assistance. If your company is interested, please contact dkipping@qualcomm.com.

The meeting serves as a forum for LLVM, Clang, LLDB and other LLVM project developers and users to get acquainted, learn how LLVM is used, and exchange ideas about LLVM and its (potential) applications. More broadly, we believe the event will be of particular interest to the following people:

- Active developers of projects in the LLVM Umbrella (LLVM core, Clang, LLDB,

LLVM-commits
LLVM-branch-commits
LLVM-testresults

IRC Channel:
irc.oftc.net #llvm

Dev. Resources:
doxygen ViewVC
Blog Bugzilla
Buildbot LNT Coverage
Scan-build llvm-cov

Release Emails

5.0.0: Sep 2017
4.0.1: Jul 2017
4.0.0: Mar 2017
3.9.1: Dec 2016
3.9.0: Sep 2016
3.8.1: July 2016
3.8.0: Mar 2016
3.7.1: Jan 2016
3.7.0: Sep 2015
3.6.2: Jul 2015
3.6.1: May 2015
3.5.2: Apr 2015
3.6: Feb 2015
3.5.1: Jan 2015
3.5: Sep 2014
3.4.2: June 2014
3.4.1: May 2014
3.4: Jan 2014
3.3: Jun 2013
3.2: Dec 2012
3.1: May 2012
3.0: Dec 2011
2.9: Apr 2011
2.8: Oct 2010
2.7: Apr 2010
2.6: Oct 2009
2.5: Mar 2009
2.4: Nov 2008

libc++, compiler_rt, klee, dragonegg, etc).
- Anyone interested in using these as part of another project.
- Compiler, programming language, and runtime enthusiasts.
- Those interested in using compiler and toolchain technology in novel and interesting ways.

We also invite you to sign up for the official Developer Meeting mailing list to be kept informed of updates concerning the meeting.

Talk Slides and Videos

Media	Talk
[Slides]	**Developer Meeting Kickoff** Chris Lattner
[Slides] [Video]	**Extending Clang** Doug Gregor, *Apple*
[Slides] [Video] (Computer) [Video] (Mobile)	**Intel OpenCL SDK Vectorizer** Nadav Rotem, *Intel*
[Video] (Computer) [Video] (Mobile)	**Clang MapReduce -- Automatic C++ Refactoring at Google Scale** Chandler Carruth, *Google*
[Slides] [Video] (Computer) [Video] (Mobile)	**PTX Back-End: GPU Programming With LLVM** Just in Holewinski, *Ohio State*
[Slides]	**Integrating LLVM into FreeBSD** Brooks Davis, *The FreeBSD Project*

2.3: Jun 2008
2.2: Feb 2008
2.1: Sep 2007
2.0: May 2007
Older News

Maintained by the
llvm-admin team

[Video] (Computer) [Video] (Mobile)	
[Slides] [Video] (Computer) [Video] (Mobile)	**Porting LLVM to a Next Generation DSP** Taylor Simpson, *QuIC*
[Slides] [Video] (Computer) [Video] (Mobile)	**DXR: Semantic Code Browsing with Clang** Joshua Cranmer, *Mozilla*
[Slides] [Video]	**LLVM MC In Practice** Jim Grosbach, Owen Anderson *Apple*
[Slides] [Video] (Computer) [Video] (Mobile)	**Using clang in the Chromium project** Nico Weber, Hans Wennborg, *Google*
[Slides] [Video] (Computer) [Video] (Mobile)	**Polly - First successful optimizations - How to proceed?** Tobias Grosser, *ENS/INRIA*
[Slides] [Video] (Computer) [Video] (Mobile)	**Android Renderscript** Stephen Hines, *Google*

[Slides] [Video] (Computer) [Video] (Mobile)	**SKIR: Just-in-Time Compilation for Parallelism with LLVM** Jeff Fifield, *University of Colorado*
[Slides] [Video]	**Register Allocation in LLVM 3.0** Jakob Olesen, *Apple*
[Slides] (PDF) [Slides] (HTML) [Video] (Computer) [Video] (Mobile)	**Exporting 3D scenes from Maya to WebGL using clang and LLVM** Jochen Wilhelmy, *consultant*
[Slides] [Video] (Computer) [Video] (Mobile)	**Super-optimizing LLVM IR** Duncan Sands, *DeepBlueCapital*
[Slides] [Video] (Computer) [Video] (Mobile)	**Finding races and memory errors with LLVM instrumentation** Konstantin Serebryany, *Google*
[Slides] (PDF) [Slides] (HTML) [Video] (Computer) [Video] (Mobile)	**Thread Safety Annotations in Clang** DeLesley Hutchins, *Google*

Talk Abstracts

Integrating LLVM into FreeBSD
Brooks Davis - The FreeBSD Project
The FreeBSD Project has been actively working to incorporate tools from the LLVM project into our base system including clang, libc++, and possibly lldb. This talk will cover our efforts so far including our plans to ship FreeBSD 9.0 with clang in the base system. I will cover both our current work to replace GPL licensed components with BSD(ish) licensed components and future or experimental work to incorporate new technologies made possible by LLVM.

Clang MapReduce -- Automatic C++ Refactoring at Google Scale
Chandler Carruth - Google
Google has over 100 million lines of code, and our biggest programming language is C++. We have a single, shared codebase developed primarily on mainline. We build every binary and all of its libraries from scratch every time, allowing us to incrementally evolve APIs and libraries over time. The entire development process is extremely incremental in nature, and even API-breaking changes are a regular occurrence. However, for core libraries used throughout the codebase, this development model is a huge challenge: how do we incrementally evolve an API in use by tens of thousands of other libraries? The answer is to use Clang to automatically refactor APIs and their users across the codebase. How do we scale Clang up to possibly the single largest unified codebase in the world? The same way Google scales anything else: MapReduce. By coupling Clang’s library design and architecture to existing Google infrastructure we can automatically compile, analyze, and refactor the entire Google codebase in minutes. In this talk, I will dive into the challenges of refactoring C++ code, how we’re using Clang and making it even better at solving them, and how we scale these solutions to the size of our codebase.

Thread Safety Annotations in Clang
DeLesley Hutchins - Google
This talk introduces the new thread safety annotations for Clang and describes the static analysis used to check them. These annotations can be used to specify properties such as whether a variable is guarded by a particular mutex, or a desired lock acquisition order. These annotations were originally introduced in gcc, and have recently been reimplemented in Clang.

Extending Clang
Doug Gregor - Apple
You have an idea for the next great C(++) language feature, but how do you realize that idea? This talk will describe how to extend Clang to add new language features, from parsing and AST-construction basics to properly handling C++ templates and ensuring smooth integration into IDEs. Particular attention will be given to capturing source information in ASTs, developing bulletproof semantic analysis, and write proper regression tests to exercise the various aspects of a language feature.

Super-optimizing LLVM IR
Duncan Sands - DeepBlueCapital / CNRS
I will describe a tool to harvest expression sequences from LLVM IR and automatically discover equivalent simplified expressions. The original version of this tool only looked for sub-expressions that were equivalent to the whole, but nonetheless discovered many simplifications missed by the LLVM optimizers (most of these have now been implemented in LLVM). The tool has since been extended to a general super optimizer by Rafael Auler.

Register Allocation in LLVM 3.0
Jakob Olesen - Apple
An overview of the features in LLVM's new register allocator

SKIR: Just-in-Time Compilation for Parallelism with LLVM

Jeff Fifield - University of Colorado

The Stream and Kernel Intermediate Representation (SKIR) is a small set of LLVM intrinsics for expressing parallel computation as a graph of sequential processes (kernels) communicating over abstract data channels (streams). At runtime, programs use the SKIR intrinsics to identify functions to use as kernels, to connect kernels together using streams, and to execute resulting program graphs. Formally, the stream parallelism expressed by SKIR programs can be viewed as a generalization of Kahn process networks. More practically, we can use SKIR as a compilation target for high level languages and frameworks containing this style of program decomposition and communication. We have used SKIR to implement a compiler for the StreamIt language, to create a C++ user library for stream/data/pipeline parallel programming, and to enable parallel programming for JavaScript applications. SKIR is implemented on top of the LLVM JIT compiler and a work stealing task scheduler. We use runtime compilation because of the runtime construction of program graphs, so that we can support dynamic optimization, and so that we can perform dynamic recompilation for heterogeneous targets. In this talk I will describe the SKIR intrinsics and programming model, and briefly describe high level language support that has been implemented for SKIR. I will present the compilation and optimization techniques used to transform the sequential LLVM+SKIR input code into concurrent code which can be executed in parallel using dynamic scheduling techniques. I will also describe how we can use LLVM JIT compilation to dynamically increase or decrease the amount of parallelism in a SKIR program depending on runtime hardware and application characteristics. Finally, I will

describe how we can further accelerate SKIR program kernels using JIT compilation, our OpenCL backend, and GPUs.

LLVM MC In Practice
Jim Grosbach, Owen Anderson - Apple
Overview of projects and new developments in the LLVM MC layer, with emphasis on the MCJIT, binary analysis, ARM integrated assembler and ARM disassembler.

Exporting 3D scenes from Maya to WebGL using clang and LLVM
Jochen Wilhelmy - Engineering consultant Wilhelmy
Modern content creation tools such as Autodesk Maya can be seen as graphical programming language. Features like animation of attributes, embedded scripts, shading networks and vertex deformations can be translated to c++, then compiled to LLVM IR and distributed to CPU and GPU. An award winning WebGL demo is shown which was produced using this approach.

DXR: Semantic Code Browsing with Clang
Joshua Cranmer - Mozilla
DXR is a source code browser which uses a clang plugin to determine information about all types, variables, and functions in a program to make reading, searching, and understanding source code easier.

PTX Back-End: GPU Programming With LLVM
Justin Holewinski - Ohio State University
In this talk, the PTX back-end for LLVM will be discussed, including its' past, present, and future. The current status of the back-end will be explored, with an emphasis on the portions of the LLVM IR instruction set and PTX intrinsics that are currently supported during code generation. This talk will also highlight the difficulties and issues that have been discovered

while writing an LLVM back-end for a virtual ISA such as PTX, such as infinite register files. Through-out the talk, examples will be provided to highlight key features of the back-end and show preliminary performance data.
In addition to back-end details, this talk will also highlight the use of Clang as a front-end for generating PTX code for NVIDIA GPUs. Through the use of Clang and the CUDA Driver API, GPGPU programs can be developed that harness the optimization power of the LLVM compiler infrastructure. Finally, the talk will conclude with an exploration of the open issues that remain in the backend, and a discussion on how the back-end can be used within larger GPGPU compiler projects.

Finding races and memory errors with LLVM instrumentation
Konstantin Serebryany - Google
We will present two dynamic testing tools based on compile-time instrumentation, both tools use the LLVM compiler.

- AddressSanitizer (ASan) finds memory bugs, such as use-after-free and out-of-bound accesses to heap and stack. This tool could be seen as a partial replacement for Valgrind and similar tools. The major advantages over Valgrind are the speed (less than 2x slowdown on average) and the ability to handle bugs related to stack and globals.
- ThreadSanitizer (TSan) finds data races. It uses the same race detection algorithm as the Valgrind-based TSan, but compile-time instrumentation allows it to be much faster (2x-4x slowdown).

We will also share our experience in deploying theses testing tools in large software projects.

Intel OpenCL SDK Vectorizer

Nadav Rotem - Intel
In this talk, we will present our OpenCL SDK and its core technology – the vectorizer compiler. We plan to present an overview of our vectorizer and discuss our experience with the LLVM compiler toolkit over the last few years. We will discuss some of our design decisions and our and plans for future features (future instruction sets, vector select, predicated instructions, etc).

Using clang in the Chromium project
Nico Weber, Hans Wennborg - Google
The Chromium project is the open-source foundation on which the Google Chrome web browser is built. It is a multi-million line codebase that uses open source libraries such as WebKit, libpng, skia, ffmpeg, and leveldb. We have been able to build Chromium using clang since October 2010, and since then we've continuously expanded what we use clang for: as continuous build compiler to catch bugs with clang's superior diagnostics, with project-specific clang plugins that enforce coding-style guidelines or find domain-specific bugs in V8, and starting with Chrome 15 as production compiler on Mac OS X. We will share our experiences using clang in a mature open-source project and cover, e.g., how clang's fabled compilation speed fares in a distributed build system with 100 parallel jobs, which parts of a browser blow up when the default compiler is changed, and which of clang's warnings catch the most bugs in practice.

Android Renderscript
Stephen Hines - Google
Renderscript is Android's advanced 3D graphics rendering and compute API. It provides a portable C99-based language with extensions to facilitate common use cases for enhancing graphics and thread level parallelism. The Renderscript compiler frontend is based on

Clang/LLVM. It emits a portable bitcode format for the actual compiled script code, as well as reflects a Java interface for developers to control the execution of the compiled bitcode. Executable machine code is then generated from this bitcode by an LLVM backend on the device. Renderscript is thus able to provide a mechanism by which Android developers can improve performance of their applications while retaining portability.
This talk focuses on the design and implementation of Renderscript using Clang/LLVM. Renderscript leverages Clang's AST to provide a Java reflection of globally visible symbols in the compilation unit. This includes both global variables as well as invocable parameterized functions. We also transform the Clang-based AST before emitting bitcode to provide support for reference-counted types that span the Renderscript/Java memory domains. Renderscript uses bitcode as a portable code format so that we can leverage other hardware architectures in addition to the CPU in the future (GPU, DSP).
Interesting facts:

- Difference in Clang source: 6 lines (all upstreamable with a bit of configuration logic)
- Difference in LLVM source: ~300 lines (possibly worth upstreaming, but some fixes for legacy JIT mode are no longer relevant to TOT)
- The frontend compiler is layered completely on top of Clang, so everything we use is sub-classed and/or recombined for our purposes in the llvm-rs-cc compiler driver.
- The backend compiler is stripped down to fit on a tablet/smartphone stack.

Porting LLVM to a Next Generation DSP
Taylor Simpson - Qualcomm Innovation Center

This talk will describe our experiences in porting LLVM to Qualcomm's latest generation DSP, Hexagon. The overall experience of porting to a new architecture will be discussed. The challenges of achieving high quality code generation for a VLIW with LLVM with also be outlined.

Polly - First successful optimizations - How to proceed?
Tobias Grosser - ENS/INRIA
Polly, the LLVM Polyhedral Optimizer, was presented one year ago. At that point, only the basic infrastructure was in place and many important parts not even started. Even though Polly is still more research than production quality, we made big improvements during the last 12 months. Polly itself moved to the LLVM infrastructure with Bugtracker, Buildbot and VCS. It can now conveniently be loaded into clang as part of clang -O3. We also implemented automatic SIMD and OpenMP code generation and we created a bridge to the external PoCC optimizer. With PoCC and Polly, we were able to compile and _optimize_ the first benchmarks fully automatically and have shown significant improvements over clang -O3. In this talk we give a detailed update about the current status and want to present ideas on how to move further. These will include both research relevant ideas like automatic OpenCL code generation as well as concepts on how to develop robust loop transformations and vectorization for mainstream use.
http://polly.grosser.es

Poster Abstracts

Parfait - A Scalable Static Bug-Checking Tool Built on LLVM
Cristina Cifuentes, Nathan Keynes, Andrew Craik, Lian Li, Nathan Hawes, Andrew Browne,

and Manuel Valdiviezo - Oracle Labs
Parfait is a static bug-checking tool for C/C++ applications built on the LLVM framework. Parfait achieves precision and scalability at the same time by employing a layered program analysis framework. In Parfait, different analyses varying in precision and runtime expense are invoked on demand to detect defects of a specific type, effectively achieving higher precision with smaller runtime overheads.
Parfait has been deployed into several development organizations within Oracle. It is being run over millions of lines of C and C++ code on a daily basis. Parfait is currently processing code written for a variety of different platforms including:

- Oracle Solaris Studio on Solaris
- Microsoft Visual C/C++ on Windows (excluding MFC headers)
- GCC on Linux
- Intel C/C++ Compiler on Linux

Despite the size and complexity of the code bases being analyzed, the Parfait false-positive rate has remained below 10%.
This poster will present the design of the Parfait tool, summarize our experience with the LLVM infrastructure, and present more comprehensive results than the preliminary results we first presented to the LLVM community at the LLVM Developers Conference in 2009.

Code verification based on attributes annotation - Implementing custom attributes check using Clang
Michael Han - Autodesk
Introduce a tool named "Hippocrates" we developed at Autodesk based on Clang's attribute system to help engineers verify functions being behaved as designed. The talk

will focus on the motivation and goal of the tool, how we hacked Clang (would be a very high level 1000 feet overview due to time limits), and some results of using the tool on our large code base like Autodesk Maya with millions of lines C++ code.

LunarGLASS: A LLVM-based shader compiler stack
Michael Ilseman - LunarG
LunarGLASS is an LLVM-based shader compiler stack. It brings a new approach to shader compilation by splitting the common shared intermediate representation (IR) into two levels; the top level is completely platform independent while the bottom level is dynamically tailorable to different families of architecture. http://www.lunarglass.org/documentation

Symbolic Testing of OpenCL Code
Peter Collingbourne - Imperial College London
The poster will describe our research on the subject of verification of OpenCL kernels using symbolic execution. We present an effective technique for crosschecking a C program against an accelerated OpenCL version, as well as a technique for detecting data races in OpenCL programs. Our techniques are implemented in KLEE-CL, a symbolic execution engine based on KLEE and KLEE-FP that supports symbolic reasoning on the equivalence between symbolic values. Our approach is to symbolically model the OpenCL environment using an OpenCL runtime library targeted to symbolic execution. Using this model we are able to run OpenCL programs symbolically, keeping track of memory accesses for the purpose of race detection. We then compare the symbolic result against the plain C program in order to detect mismatches between the two versions. We applied KLEE-CL to the Parboil benchmark suite, the Bullet physics library and the OP2 library, in which we were able to find a

total of seven errors: three mismatches between the OpenCL and C implementations, two memory errors, one OpenCL compiler bug and one race condition.