

The Many Faces of Instrumentation: Debugging and Better Performance using LLVM in HPC

- ✓ What are LLVM, Clang, and Flang?
- ✓ How is LLVM Being Improved for HPC.
- ✓ What Facilities for Tooling Exist in LLVM?
- ✓ Opportunities for the Future!

Protools 2019 @ SC19
2019-11-17

Hal Finkel
Leadership Computing Facility
Argonne National Laboratory

hfinkel@anl.gov

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 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.

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](#)

5/27/2014 The bgclang Compiler | Argonne Leadership Computing Facility

Content to | [Clang](#) | [LLVM](#) | [User Support](#)

Argonne Leadership Computing Facility
An Office of Science user facility

ABOUT | RESOURCES & SUPPORT | SCIENCE AT ALCF | NEWS & EVENTS | USER SERVICES | GETTING STARTED

User Guides

- How to Get an Allocation
- New User Guide
- Accounts & Access
- Allocations
- Mira/Cetus/Vesta
 - System Overview
 - Data Storage & File Systems
- Compiling & Linking
 - Overview of How to Compile and Link
 - Example Programs and Makefiles for BG/Q
 - How to Manage Threading
 - bgclang Compiler
 - Compiling and Linking FAQ
- Overcoming & Running Jobs
- Data Transfer
- Debugging & Profiling
- Performance Tools & APIs
- Software & Libraries
- IBM References
- Tutorials
- Publications

The bgclang Compiler

Using bgclang on Vesta, Mira and Cetus

If you have access to ALCF's Vesta, Mira and Cetus systems, this BG/Q is installed for you. You can use the software keys:

```
module load bgclang
module load bgclang-legacy
```

to have the corresponding MPI wrappers added to your path.

Other BG/Q systems

If you are working on a non-ALCF BG/Q system, and would like to install the bgclang project page: <http://trac.anl.gov/projects/llvm-bgc>

MPI and other wrappers

On an ALCF system (or any other system with a similar setup), the MPI wrappers can be easily added to your PATH (see the description of the AL wrappers are:

```
module load mpicc
module load mpicxx
module load mpicxx11
```

To use bgclang without using the MPI wrappers:

```
bgclang (or powerpc64-ibm-linux-clang) - The C99 compiler
bgclang++ (or powerpc64-ibm-linux-clang++) - The C++ compiler
bgclang++11 (or powerpc64-ibm-linux-clang++11) - The C++11 compiler
```

Mailing list and support

ALCF users may e-mail support for help with bgclang-related questions. A list to subscribe to the mailing list: <http://lists.anl.gov/mailman/listinfo/llvm-bgc-discuss>. On non-ALCF systems should use the mailing list to receive help with bgclang.

General usage

bgclang command-line argument handling is designed to be similar to gcc

<https://www.anl.gov/user-guides/bgclang-compiler>

bgclang (LLVM/clang on the BG/Q)

For usage information, and information specific to using bgclang on ALCF's BG/Q machines (Vesta, Mira and Cetus), please visit: <http://www.anl.gov/user-guides/bgclang-compiler>

Other BG/Q systems

If your system administrators have not been kind enough to install bgclang on your system, you can either direct them to this page, or install the distribution yourself. RPMs are provided (see below), and these are "relocatable" RPMs, meaning that they can be installed by a non-root user in any directory.

Please note that, if you wish to use dynamic linking (which you must do when certain features, like address sanitizer, are enabled), you must install bgclang in a directory that is mounted from the compute nodes (read-only is sufficient).

Mailing List

If you're using bgclang, please subscribe to the mailing list: <http://lists.anl.gov/mailman/listinfo/llvm-bgc-discuss>.

bgclang downloads (for installing on your own)

For those managing their own installs, note that the MPI wrappers are installed in the PREFIX/mpir (bgclang, bgclang-legacy)/bin directories. The non-MPI compiler wrappers are located in the PREFIX/bin directory.

RPMs, etc.

r209570-20140527

<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-binutils-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-sleef-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-libc++-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-stage1-ibomp-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-compiler-rt-r209570-20140527-1-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-stage1-3.4-1.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/bgclang-stage1-3.4-2.ppc64.rpm>
<http://www.mcs.anl.gov/~hfinkel/bgclang/RPMS/ppc64/vpk-g-bin-sh-1-1.ppc64.rpm>

A non-root (regular) user can install these RPMs (because they are relocatable), but in addition to specifying the installation prefix (with the --prefix argument), an alternate RPM database directory needs to be specified (in a directory to which you actually have write permission). For example, to install bgclang into the /tmp/bgclang directory using /tmp/bgclang/rpm as the RPM

<https://trac.anl.gov/projects/llvm-bgc/wiki/Install>

1/20



Clang, LLVM, etc.

- ✓ LLVM is a liberally-licensed(*) infrastructure for creating compilers, other toolchain components, and JIT compilation engines.
- ✓ Clang is a modern C++ frontend for LLVM
- ✓ LLVM and Clang will play significant roles in exascale computing systems!

(*) Now under the Apache 2 license with the LLVM Exception

LLVM/Clang is both a research platform and a production-quality compiler.

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](#)

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.

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.

LLVM can be used as a [research project](#) at the [University of Illinois](#), with the goal of a modern, SSA-based compilation platform of supporting both static and dynamic compilation of arbitrary languages. Since then, LLVM to be an umbrella project of a number of subprojects, many are being used in production by a variety of [commercial and open source](#) well as being widely used in [research](#). Code in the LLVM is licensed under the ["UIUC" BSD](#).

LLVM sub-projects of LLVM are:

LLVM Core libraries provide a common source- and target-independent [optimizer](#), along with [generation support](#) for many other CPUs (as well as some less common ones). These libraries are [around a well specified code generation](#) known as the LLVM intermediate representation ("LLVM"). The LLVM Core libraries are [documented](#), and it is particularly [easy to invent your own language](#) (or an existing compiler) to use [LLVM as an optimizer and code generator](#).

[LLVM](#) is an "LLVM native" ++Objective-C compiler, which

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

Forward 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](#)

5/27/2014

llvm-bgg

bgclang (LLVM/clang on the BG/Q)

For usage information, and information specific to using bgclang on ALCF's BG/Q machines (Vesta, Mira and Cetus), please visit: <http://www.alcf.anl.gov/user-guides/bgclang-compiler>

Other BG/Q systems

If your system administrators have not been kind enough to install bgclang on your system, you can either direct them to this page, or install the distribution yourself. RPMs are provided (see below), and these are *relocatable* RPMs, meaning that they can be installed by a non-root user in any directory.

Please note that, if you wish to use dynamic linking (which you must do when certain features, like address sanitizer, are enabled), you must install bgclang in a directory that is mounted from the compute nodes (read-only is sufficient).

Mailing List

If you're using bgclang, please subscribe to the mailing list: <http://lists.alcf.anl.gov/mailman/listinfo/llvm-bgg-discuss>.

bgclang downloads (for installing on your own)

For those managing their own installs, note that the MPI wrappers are installed in the PREFIX/mpir/bgclang/bgclang.legacy/bin directories. The non-MPI compiler wrappers are located in the PREFIX/w/bin directory.

RPMs, etc.

r209570-20140527

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-binutils-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-sleefer-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-libcxx-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-libomp-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-compiler-rt-r209570-20140527-1-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-stage1-3.4-1.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/bgclang-stage1-libcxx-3.4-2.ppc64.rpm>

<http://www.mcs.anl.gov/~hfinke/bgclang/RPMS/ppc64/vpk-g-bin-sh-1-1.ppc64.rpm>

A non-root (regular) user can install these RPMs (because they are relocatable), but in addition to specifying the installation prefix (with the --prefix argument), an alternate RPM database directory needs to be specified (in a directory to which you actually have write permission). For example, to install bgclang into the /tmp/bgclang directory using /tmp/bgclang/rpm as the RPM

<http://www.alcf.anl.gov/projects/llvm-bgg/wiki/Start>

3/20

5/27/2014 The bgclang Compiler | Argonne Leadership Computing Facility

Content to: [Cetus] [Vesta] [Mira] [Cetus] [User Support]

Argonne Leadership Computing Facility
an Office of Science user facility

ABOUT | RESOURCES & SUPPORT | SCIENCE AT ALCF | NEWS & EVENTS | USER SERVICES | GETTING STARTED

User Guides

- How to Get an Allocation
- New User Guide
- Accounts & Access
- Allocations
- Mira/Cetus/Vesta
 - System Overview
 - Data Storage & File Systems
- Compiling & Linking
 - Overviews of How to Compile and Link
 - Example Programs and Makefiles for BG/Q
 - How to Manage Threading
 - bgclang Compiler
 - Compiling and Linking FAQ
- Overseeing & Running Jobs
 - Data Transfer
 - Debugging & Profiling
 - Performance Tools & APIs
 - Software & Libraries
 - IBM References
- Tools
- Policies

The bgclang Compiler

Using bgclang on Vesta, Mira and Cetus

If you have access to ALCF's Vesta, Mira and Cetus systems, the BG/Q-enhanced LLVM is installed for you. You can use the software keys:

Package	Location
~mcsanl-llvm-bgg	bgclang wrappers and toolchain
~mcsanl-llvm-bgg-legacy	bgclang legacy wrappers and toolchain

to have the corresponding MPI wrappers added to your path.

Other BG/Q systems

If you are working on a non-ALCF BG/Q system, and would like to install the bgclang compiler, please see the bgclang project page: <http://www.alcf.anl.gov/projects/llvm-bgg>

MPI and other wrappers

If you are working on a non-ALCF BG/Q system, the MPI wrappers and programs can be easily added to your PATH (see the description of the ALCF system wrappers are:

Wrapper	Compiler
mcsanl-llvm-bgg	The MPI C++ compiler
mcsanl-llvm-bgg-legacy	The MPI C++ compiler
mcsanl-llvm-bgg-legacy	The MPI C++ compiler

To use bgclang without using the MPI wrappers:

Compiler	Wrapper
bgclang (or powerpc64-llvm-clang)	The C++ compiler
bgclang++ (or powerpc64-llvm-clang++)	The C++ compiler
bgclang++ (or powerpc64-llvm-clang++)	The C++ compiler
bgclang++ (or powerpc64-llvm-clang++)	The C++ compiler

Documentation Feedback

Please provide feedback to help guide us as we continue to build documentation for our new computing resource.

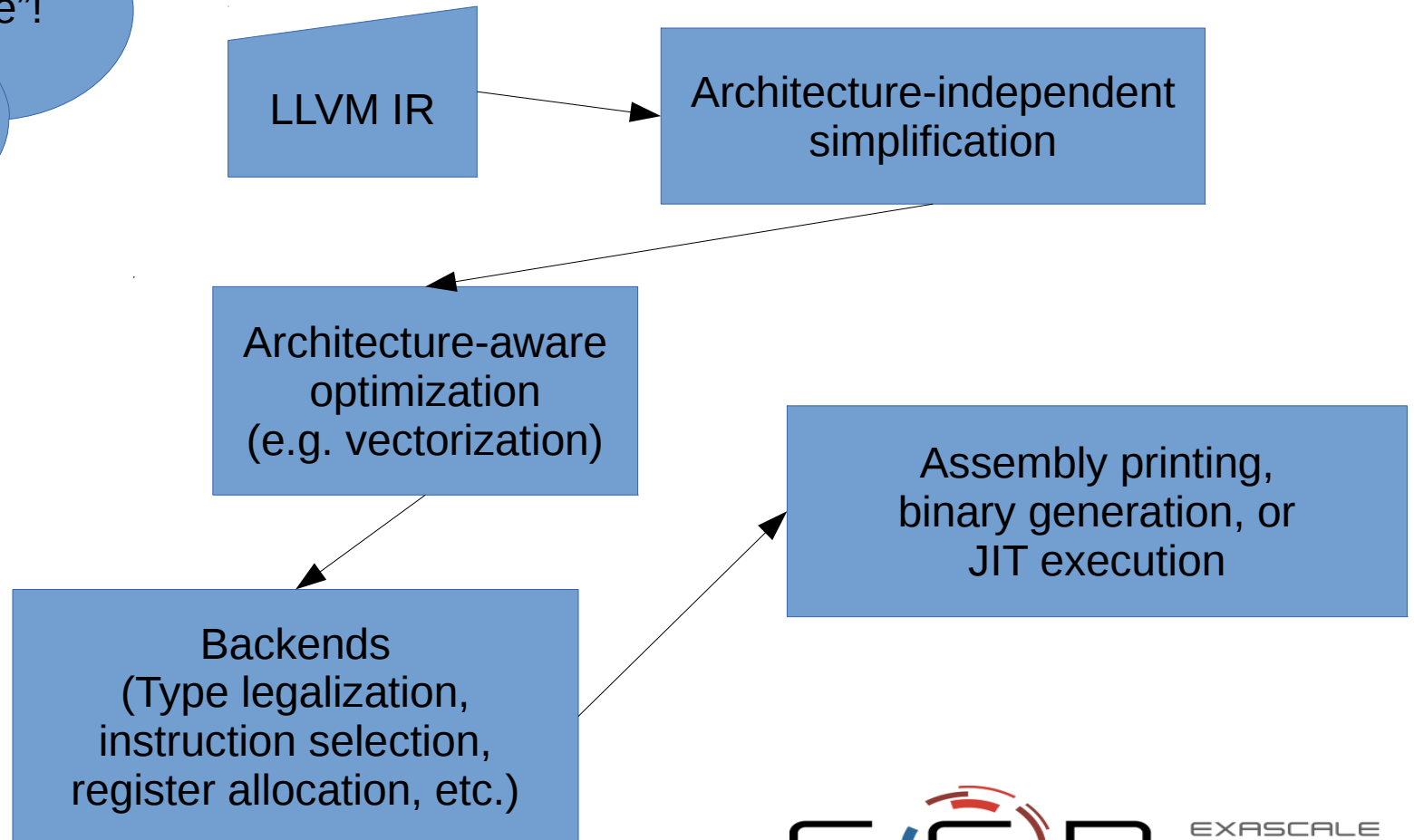
[Feedback Form]

<https://www.alcf.anl.gov/user-guides/bgclang-compiler>

What is LLVM:

LLVM is **not** a “low-level virtual machine”!

LLVM is a multi-architecture infrastructure for constructing compilers and other toolchain components.



What is Clang:

Clang is a C++ frontend for LLVM...



- For basic compilation, Clang works just like gcc – using clang instead of gcc, or clang++ instead of g++, in your makefile will likely “just work.”
- Clang has a scalable LTO, check out: <https://clang.llvm.org/docs/ThinLTO.html>

5/28/2014 "clang" C Language Family Frontend for LLVM

clang: a C language family frontend for LLVM

The goal of the Clang project is to create a new C, C++, Objective C and Objective C++ frontend for the LLVM compiler. You can [get](#) and [build](#) the source today.

Features and Goals

Some of the goals for the project include the following:

End-User Features:

- Fast compile and low memory use
- Expressive diagnostics ([examples](#))
- GCC compatibility

Utility and Applications:

- Modular library based architecture
- Support diverse clients (refactoring, static analysis, code generation, etc.)
- Allow tight integration with IDEs
- Use the LLVM BSD License

Internal Design and Implementation:

- A realworld, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

Of course this is only a rough outline of the goals and features of Clang. To get a true sense of what it is all about, see the [Features](#) section, which breaks each of these down and explains them in more detail.

Why?

Development of the new front-end was started out of a need for a compiler that allows better diagnostics, better integration with IDEs, a license that is compatible with commercial products, and a nimble compiler that is easy to develop and maintain. All of these were motivations for starting work on a

<http://clang.llvm.org>

5/28/2014 Clang Static Analyzer

Clang Static Analyzer

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs.

Currently it can be run either as a [standalone tool](#) or [within code](#). The standalone tool is invoked from the command line, and is intended to be run on code with a build of a code base.

The analyzer is 100% open source and is part of the Clang project. Like the rest of Clang, the analyzer is implemented as a C++ library that can be used by other tools and applications.

Download

Mac OS X

- Latest build (only binary, 12.5k)
- [Download Clang Static Analyzer \(Swift binary 15.2k\)](#)
- [Release notes](#)
- This build can be used both from the command line and from within Xcode
- [Installation](#) and [usage](#)

Other Platforms

For other platforms, please follow the instructions for [building the analyzer](#) from source code.

What is Static Analysis?

The term "static analysis" is confusing, but here we use it to mean a collection of algorithms and techniques used to analyze source code in order to automatically find bugs. The idea is similar to what is done by code coverage (which can be used for finding coding errors) but it takes that idea a step further and finds bugs that are traditionally found using expensive debugging techniques such as testing.

Static analysis bug-finding tools have evolved over the last several decades from basic syntactic checkers to those that find deep bugs by reasoning about the semantics of code. The goal of the Clang Static Analyzer is to provide a high-quality static analysis framework for analyzing C, C++, and Objective-C programs that is freely available, extensible, and has a high quality of implementation.

Part of Clang and LLVM

As its name implies, the Clang Static Analyzer is built on top of Clang and LLVM. Strictly speaking, the analyzer is part of Clang, as Clang consists of a set of reusable C++ libraries for building powerful compiler tools. The static analysis engine used by the Clang Static Analyzer is a Clang library, and has the capability to be used in different contexts and by different tools.

Important Points to Consider

While we believe that the static analyzer is already very useful for finding bugs, we ask you to bear in mind a few points when using it.

Work in Progress

The analyzer is a continuous work in progress. There are many planned enhancements to improve both the precision and scope of its analysis algorithms as well as the kinds of bugs it will find. While there are fundamental limitations to what static analysis can do, we have a long way to go before hitting that wall.

<http://clang-analyzer.llvm.org>

The primary sub-projects of LLVM are:

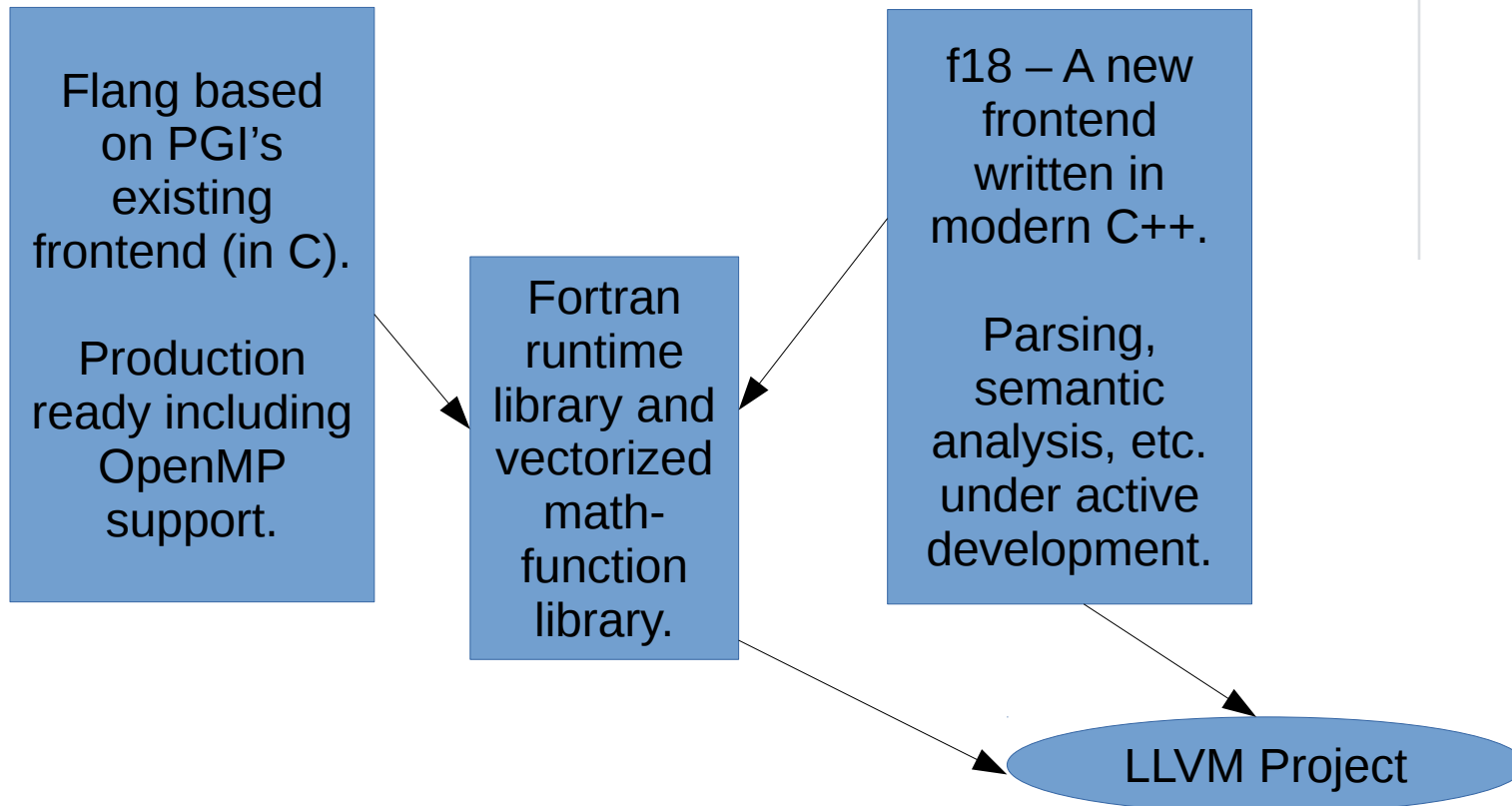
1. The **LLVM Core** libraries provide a modern source- and target-independent [optimizer](#), along with [code generation support](#) for many popular CPUs (as well as some less common ones!) These libraries are built around a [well specified](#) code representation known as the LLVM intermediate representation ("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 aims to deliver amazingly fast compiles (e.g. about [3x faster than GCC](#) when compiling Objective-C code in a debug configuration), extremely useful [error and warning messages](#) and to provide a platform for building great source level tools. The [Clang Static Analyzer](#) is a tool that automatically finds bugs in your code, and is a great example of the sort of tool that can be built using the Clang frontend as a library to parse C/C++ code.
3. The **LLDB** project builds on libraries provided by LLVM and Clang to provide a great native debugger. It uses the Clang ASTs and expressions, LLVM JIT, LLVM disassembler, etc so that it provides an experience that "just works". It is also blazing fast and much more memory efficient than GDB at loading symbols.
4. The [libc++](#) and [libc++ ABI](#) projects provide a standard conformant and high-performance implementation of the C++ Standard Library, giving full support for C++11.
5. The [compiler-rt](#) project provides highly tuned implementations of the low-level code generator support routines like [__cxa_guard_acquire](#) and [__cxa_guard_release](#) that are generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It also provides implementations of run-time libraries for dynamic testing tools such as [AddressSanitizer](#), [ThreadSanitizer](#), [MemorySanitizer](#), and [DataFlowSanitizer](#).
6. The **OpenMP** subproject provides an [OpenMP](#) runtime for use with the OpenMP implementation in Clang.
7. The **polly** project implements a suite of cache-locality optimizations as well as auto-parallelism and vectorization.
8. The **libclc** project aims to implement the OpenCL standard library.

The core LLVM compiler-infrastructure components are one of the subprojects in the LLVM project. These components are also referred to as "LLVM."

11. The **lld** project aims to be the built-in linker for clang/llvm. Currently, clang must invoke the system linker to produce executables.

What About Flang?

- Started as a collaboration between DOE and NVIDIA/PGI. Now also involves ARM and other vendors.
- Flang (f18+runtime) has been accepted to become a part of the LLVM project.
- Two development paths:



<https://github.com/flang-compiler/flang>

README.md

Flang

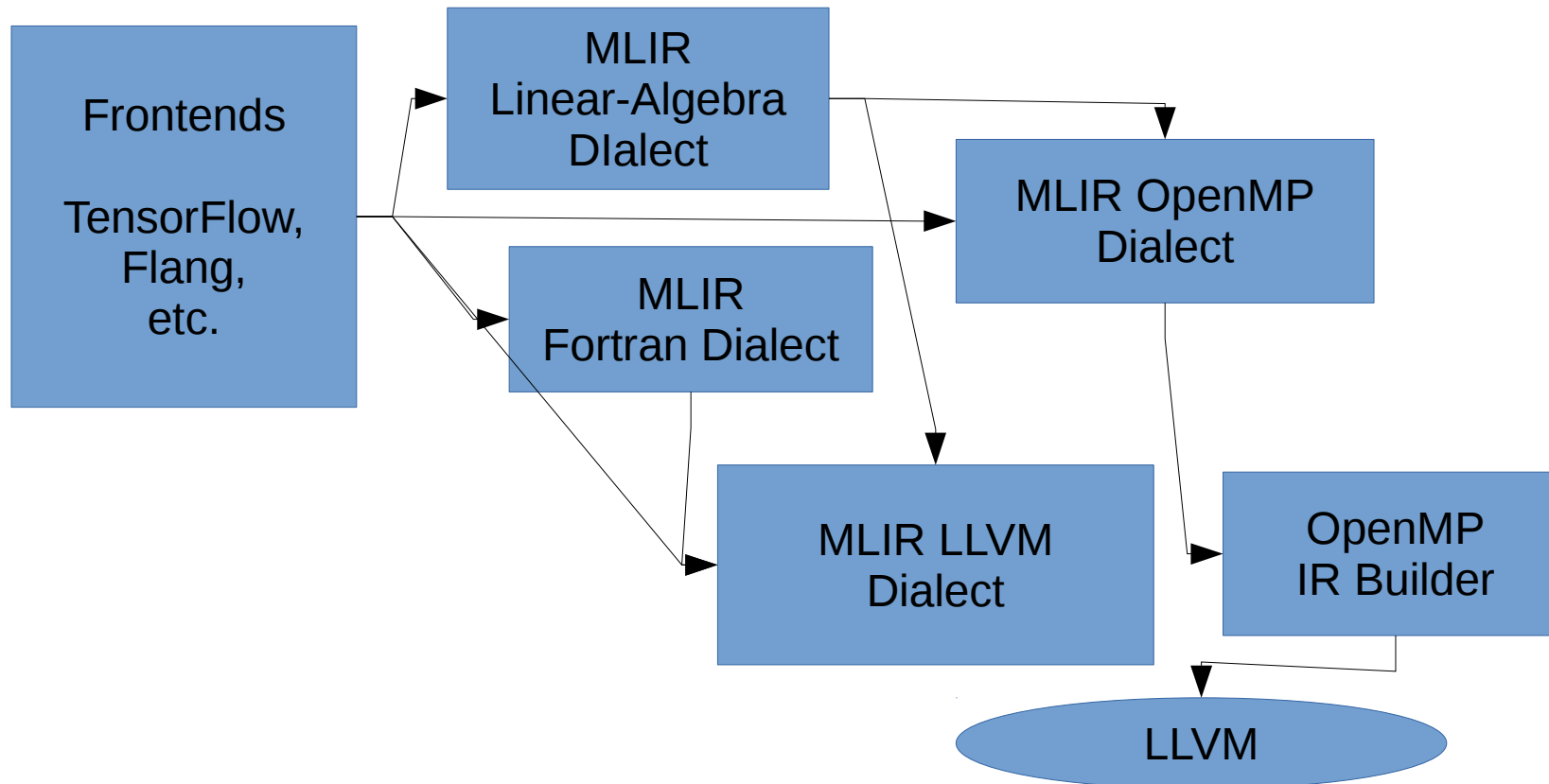
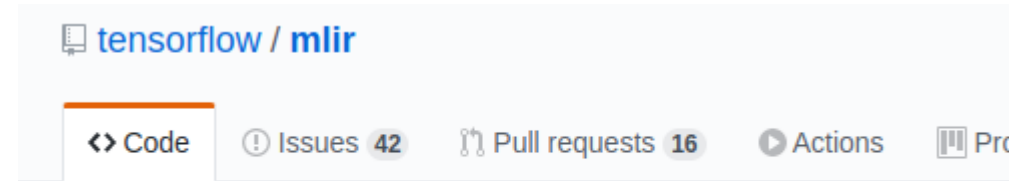
Flang is a Fortran compiler targeting LLVM.

Visit the flang wiki for more information:

<https://github.com/flang-compiler/flang/wiki>

What About MLIR?

- Started as a part of Google's TensorFlow project.
- MLIR will become part of the LLVM project.
- MLIR is built around the simultaneous support of multiple dialects.



Clang Can Compile CUDA!

- CUDA is the language used to compile code for NVIDIA GPUs.
- Support now also developed by AMD as part of their HIP project.

```
$ clang++ axpy.cu -o axpy --cuda-gpu-arch=<GPU arch>
```

For example:
--cuda-gpu-arch=sm_35

When compiling, you may also need to pass --cuda-path=/path/to/cuda if you didn't install the CUDA SDK into /usr/local/cuda (or a few other "standard" locations).

For more information, see: <http://llvm.org/docs/CompileCudaWithLLVM.html>

Clang's CUDA aims to provide better support for modern C++ than NVIDIA's nvcc.



Existing LLVM Capabilities

- Clang Static Analysis (including now integration with the Z3 SMT solver)
- Clang Warnings and Provided-by-Default Analysis (e.g., MPI-specific warning messages)
- LLVM-based static analysis (using, e.g., optimization remarks)
- LLVM instrumentation-based checking (e.g., UBSan)
- LLVM instrumentation-based checking using Sanitizer libraries (e.g., AddressSanitizer)
- Lightweight instrumentation for performance collection (e.g., Xray)
- Low-level performance analysis (e.g., llvm-mca)

MPI-specific warning messages

```
mpit2.c:18:17: warning: argument type 'char *' doesn't match specified 'MPI' type tag that requires 'double *' [-Wtype-safety]
    rc = MPI_Send(&outmsg, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
                   ^~~~~~ ~~~~~~
```

These are not really MPI specific, but uses the “type safety” attributes inspired by this use case:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype)
```

```
    __attribute__(( pointer_with_type_tag(mpi,1,3) ));
```

```
...
```

```
#define MPI_DATATYPE_NULL ((MPI_Datatype) 0xa0000000)
```

```
#define MPI_FLOAT          ((MPI_Datatype) 0xa0000001)
```

```
...
```

```
static const MPI_Datatype mpich_mpi_datatype_null __attribute__(( type_tag_for_datatype(mpi,void,must_be_null) )) = 0xa0000000;
```

```
static const MPI_Datatype mpich_mpi_float          __attribute__(( type_tag_for_datatype(mpi,float) ))          = 0xa0000001;
```

See Clang's test/Sema/warn-type-safety-mpi-hdf5.c, test/Sema/warn-type-safety.c and

test/Sema/warn-type-safety.cpp for more examples,

and: <http://clang.llvm.org/docs/AttributeReference.html#type-safety-checking>

Optimization Reporting - Design Goals

To get information from the backend (LLVM) to the frontend (Clang, etc.)

- ✓ To enable the backend to generate diagnostics and informational messages for display to users.
- ✓ To enable these messages to carry additional “metadata” for use by knowledgeable frontends/tools
- ✓ To enable the programmatic use of these messages by tools (auto-tuners, etc.)
- ✓ To enable plugins to generate their own unique messages

```
sqlite3.c:60198:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
    if( ^sqlite3StrICmp(zLeft, "case_sensitive_like")==0 ){
```

```
sqlite3.c:60200:40: remark: getBoolean inlined into sqlite3Pragma [-Rpass=inline]
    sqlite3RegisterLikeFunctions(db, ^getBoolean(zRight));
```

```
sqlite3.c:60213:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
    if( ^sqlite3StrICmp(zLeft, "integrity_check")==0
```

```
sqlite3.c:60214:7: remark: sqlite3StrICmp inlined into sqlite3Pragma [-Rpass=inline]
    || ^sqlite3StrICmp(zLeft, "quick_check")==0
```

```
sqlite3.c:44776:8: remark: sqlite3VdbeMemFinalize inlined into sqlite3VdbeExec [-Rpass=inline]
    rc = ^sqlite3VdbeMemFinalize(pMem, pOp->p4.pFunc);
```

See also: <http://llvm.org/docs/Vectorizers.html#diagnostics>

Sanitizers

The sanitizers (some now also supported by GCC) – Instrumentation-based debugging

- Checks get compiled in (and optimized along with the rest of the code) – Execution speed an order of magnitude or more faster than Valgrind
- You need to choose which checks to run at compile time:
 - Address sanitizer: -fsanitize=address – Checks for out-of-bounds memory access, use after free, etc.: <http://clang.llvm.org/docs/AddressSanitizer.html>
 - Leak sanitizer: Checks for memory leaks; really part of the address sanitizer, but can be enabled in a mode just to detect leaks with -fsanitize=leak: <http://clang.llvm.org/docs/LeakSanitizer.html>
 - Memory sanitizer: -fsanitize=memory – Checks for use of uninitialized memory: <http://clang.llvm.org/docs/MemorySanitizer.html>
 - Thread sanitizer: -fsanitize=thread – Checks for race conditions: <http://clang.llvm.org/docs/ThreadSanitizer.html>
 - Undefined-behavior sanitizer: -fsanitize=undefined – Checks for the execution of undefined behavior: <http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
 - Efficiency sanitizer [Recent development]: -fsanitize=efficiency-cache-frag, -fsanitize=efficiency-working-set (-fsanitize=efficiency-all to get both)

And there's more, check out <http://clang.llvm.org/docs/> and Clang's include/clang/Basic/Sanitizers.def for more information.

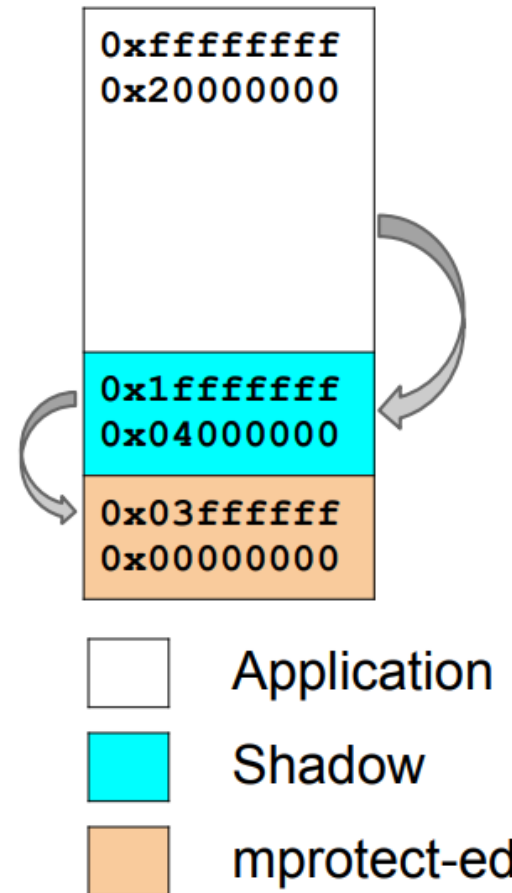
Address Sanitizer

```
int main(int argc, char **argv) {  
    int *array = new int[100];  
    delete [] array;  
    return array[argc]; } // BOOM  
% clang++ -O1 -fsanitize=address a.cc && ./a.out  
==30226== ERROR: AddressSanitizer heap-use-after-free  
READ of size 4 at 0x7faa07fce084 thread T0  
    #0 0x40433c in main a.cc:4  
0x7faa07fce084 is located 4 bytes inside of 400-byte region  
freed by thread T0 here:  
    #0 0x4058fd in operator delete[](void*) _asan_rtl_  
    #1 0x404303 in main a.cc:3  
previously allocated by thread T0 here:  
    #0 0x405579 in operator new[](unsigned long) _asan_rtl_  
    #1 0x4042f3 in main a.cc:2
```

http://www.llvm.org/devmtg/2012-11/Serebryany_TSan-MSan.pdf

ASan shadow memory

Virtual address space



Instrumentation

```
*a = ...  
  
↓  
  
char *shadow  
    = addr >> 3;  
if (*shadow)  
    ReportError(a);  
*a = ...
```


Thread Sanitizer

```
#include <thread>

int g_i = 0;
std::mutex g_i_mutex; // protects g_i

void safe_increment()
{
    // std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;
}

int main()
{
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();
}
```

Everything is fine if I uncomment
this line...

Thread Sanitizer

```
$ clang++ -std=c++11 -stdlib=libc++ -fsanitize=thread -O1 -o /tmp/r1 /tmp/r1.cpp
$ /tmp/r1
```

```
=====
WARNING: ThreadSanitizer: data race (pid=486)
  Write of size 4 at 0x000001521cb8 by thread T2:
    #0 safe_increment() <null> (r1+0x00000049d2ac)
    #1 void* std::__1::__thread_proxy<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct, std::__1::default_delete<std::__1::__thread_struct>>, void (*)()>>(void*) <null> (r1+0x00000049d455)

  Previous write of size 4 at 0x000001521cb8 by thread T1:
    #0 safe_increment() <null> (r1+0x00000049d2ac)
    #1 void* std::__1::__thread_proxy<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct, std::__1::default_delete<std::__1::__thread_struct>>, void (*)()>>(void*) <null> (r1+0x00000049d455)

  Location is global '<null>' at 0x000000000000 (r1+0x000001521cb8)

  Thread T2 (tid=489, running) created by main thread at:
    #0 pthread_create /home/hfinkel/public/src/llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
    #1 std::__1::thread::thread<void (&()), , void>(void (&())) <null> (r1+0x00000049d3b6)
    #2 main <null> (r1+0x00000049d2ea)

  Thread T1 (tid=488, finished) created by main thread at:
    #0 pthread_create /home/hfinkel/public/src/llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:902 (r1+0x000000420aa5)
    #1 std::__1::thread::thread<void (&()), , void>(void (&())) <null> (r1+0x00000049d3b6)
    #2 main <null> (r1+0x00000049d2dd)

SUMMARY: ThreadSanitizer: data race (/tmp/r1+0x49d2ac) in safe_increment()
=====
ThreadSanitizer: reported 1 warnings
```

LLVM XRay

Lightweight instrumentation library, add places to patch in instrumentation (generally to functions larger than some threshold):

```
local_block_sled_0:
    jmp . + 0x09
    (9 bytes worth of nops)
... # function prologue starts, followed by the body.
... # function epilogue starts, just before ret...
local_block_sled_1:
    retq
    (10 bytes worth of nops)
```

Can be extended to do many things, but comes with an “Flight Data-Recorder” Mode:

```
// Patch the sleds, if we haven't yet.
auto patch_status = __xray_patch();

// Maybe handle the patch_status errors.

// When we want to flush the log, we need to finalize it first, to give
// threads a chance to return buffers to the queue.
auto finalize_status = __xray_log_finalize();
if (finalize_status != XRAY_LOG_FINALIZED) {
    // maybe retry, or bail out.
}

// At this point, we are sure that the log is finalized, so we may try
// flushing the log.
auto flush_status = __xray_log_flushLog();
if (flush_status != XRAY_LOG_FLUSHED) {
    // maybe retry, or bail out.
}
```

<https://llvm.org/docs/XRay.html>

LLVM MCA

Using LLVM's instruction-scheduling infrastructure to analyze programs...

Below is an example of -bottleneck-analysis output generated by **llvm-mca** for 500 iterations of the dot-product example on btver2.

```
Cycles with backend pressure increase [ 48.07% ]
Throughput Bottlenecks:
  Resource Pressure      [ 47.77% ]
  - JFPA [ 47.77% ]
  - JFPU0 [ 47.77% ]
  Data Dependencies: [ 0.30% ]
  - Register Dependencies [ 0.30% ]
  - Memory Dependencies [ 0.00% ]

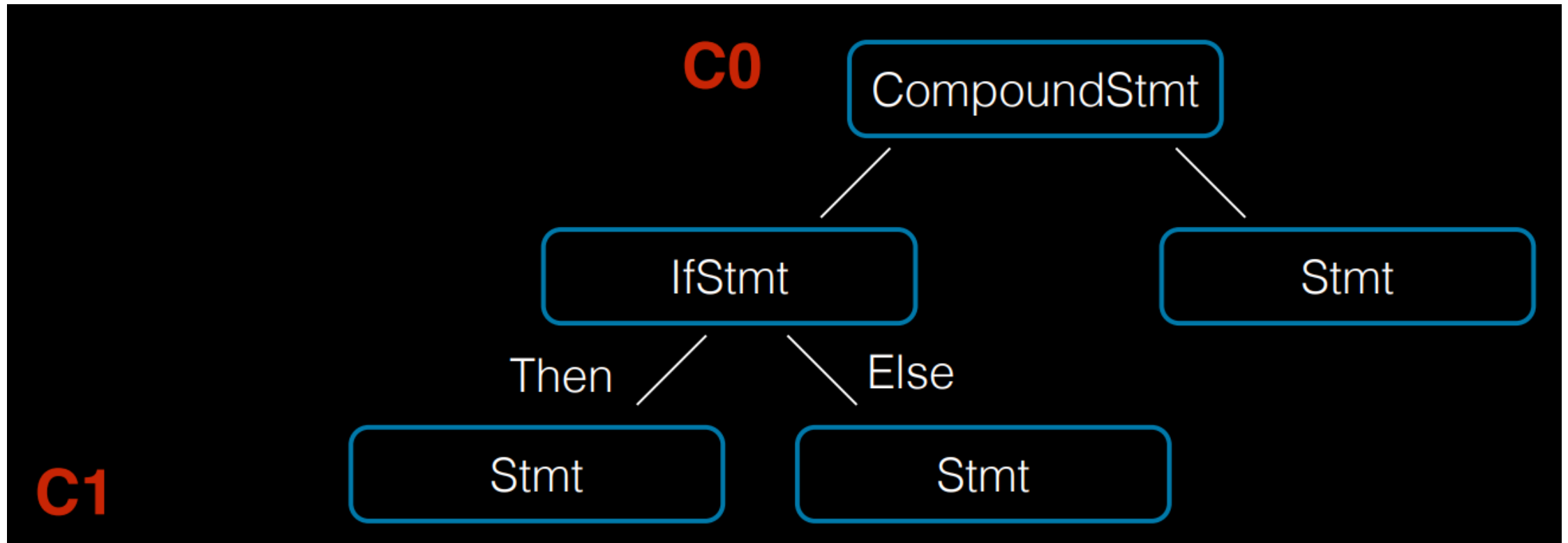
Critical sequence based on the simulation:

+----< 2.      Instruction
|              vhaddps %xmm3, %xmm3, %xmm4
|              < loop carried >
|              0.      vmulps %xmm0, %xmm1, %xmm2
+----> 1.      vhaddps %xmm2, %xmm2, %xmm3      ## RESOURCE interference: JFPA [ probability: 74% ]
+----> 2.      vhaddps %xmm3, %xmm3, %xmm4      ## REGISTER dependency: %xmm3
|              < loop carried >
+----> 1.      vhaddps %xmm2, %xmm2, %xmm3      ## RESOURCE interference: JFPA [ probability: 74% ]
```

<https://llvm.org/docs/CommandGuide/llvm-mca.html>

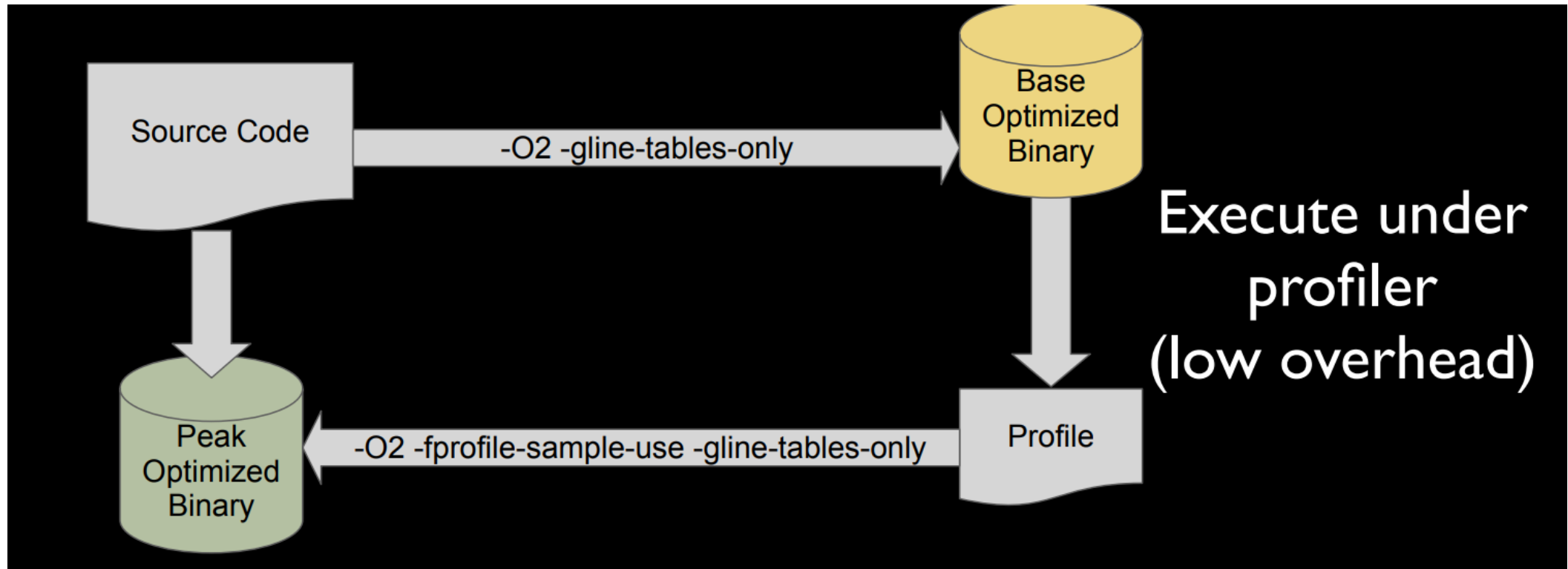
Profile-Guided Optimization

Instrumentation vs. Sampling PGO; for instrumentation:

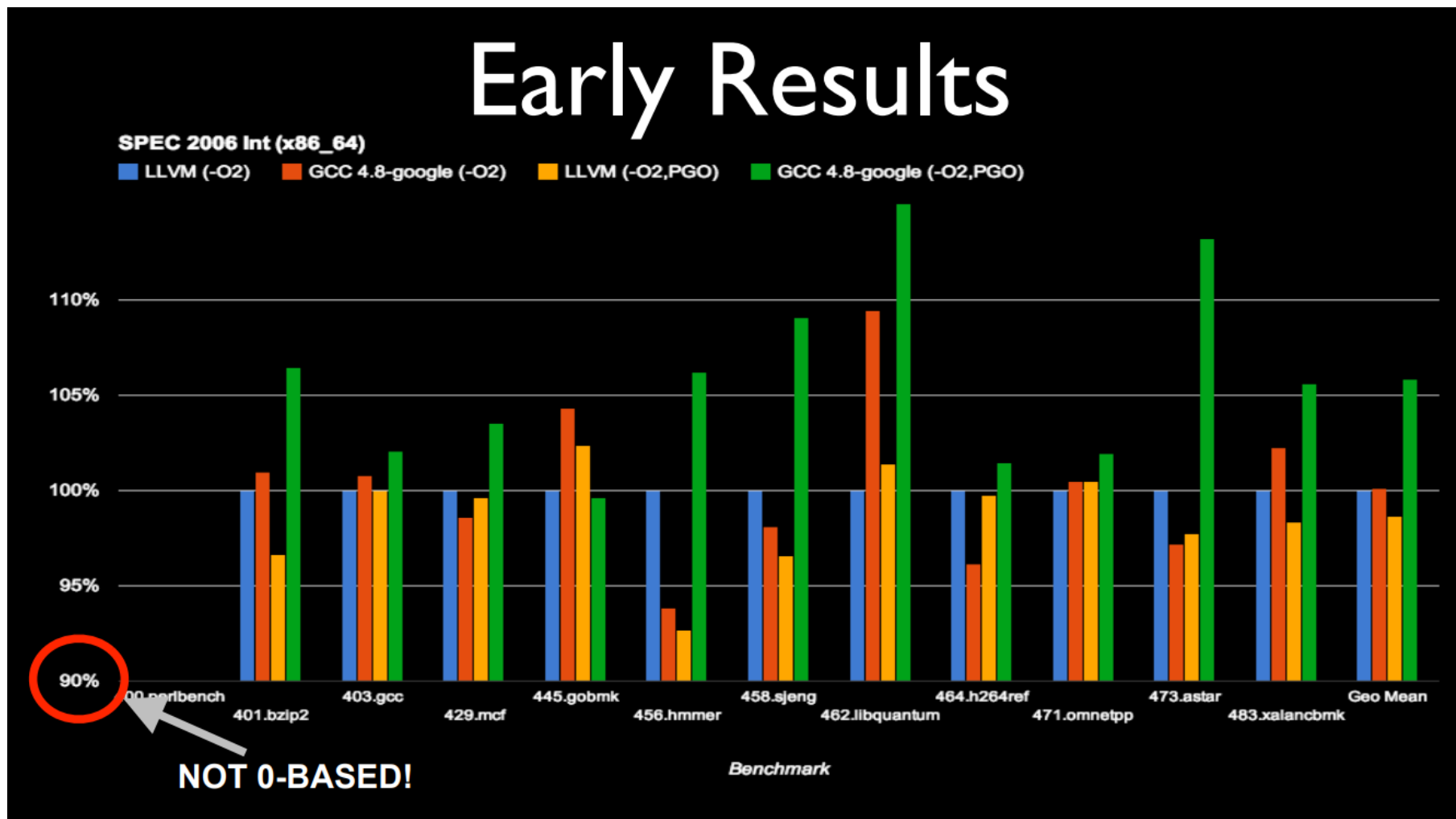


PGO

Instrumentation vs. Sampling PGO; for sampling:



PGO



<https://llvm.org/devmtg/2013-11/slides/Carruth-PGO.pdf>

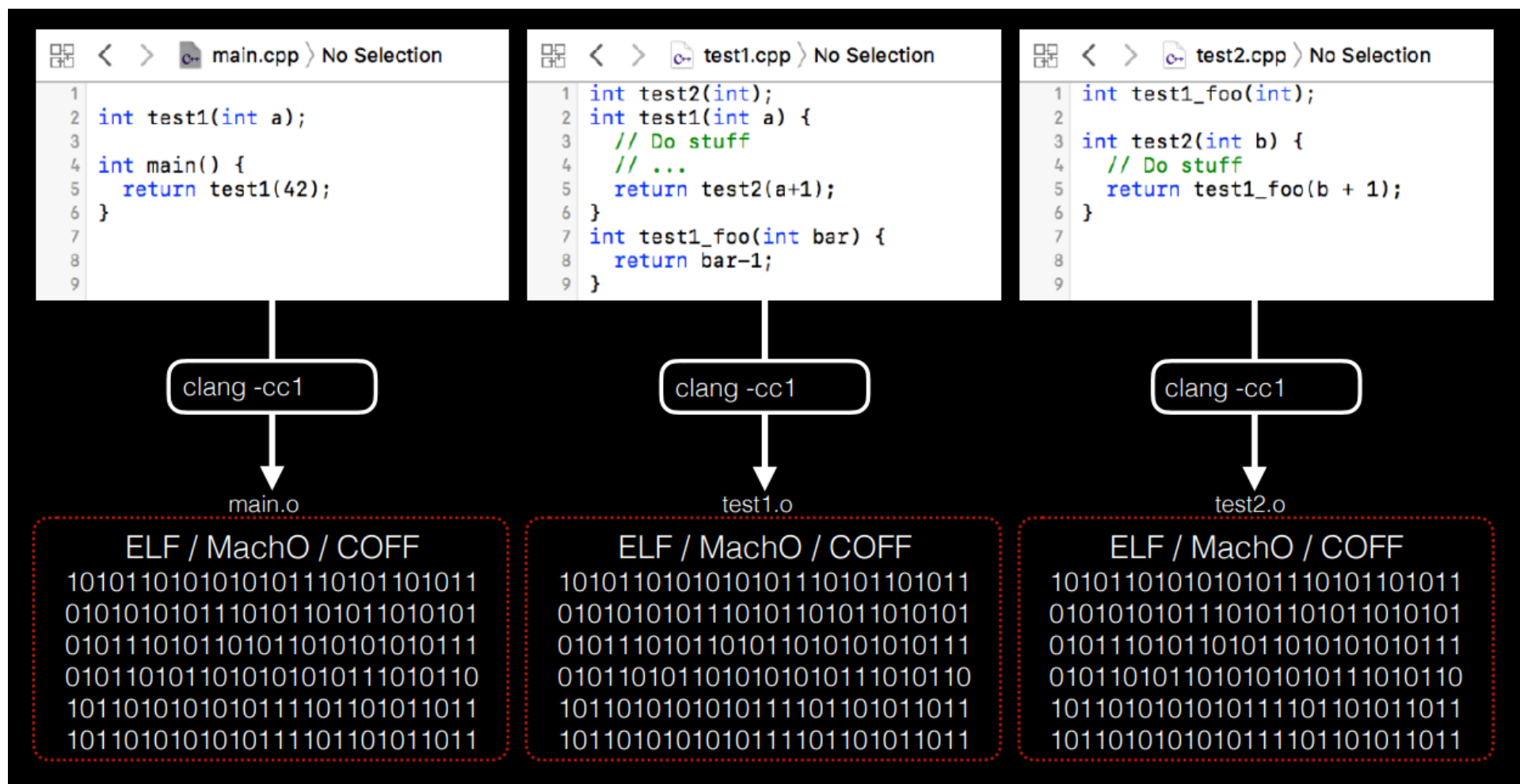


EXASCALE
COMPUTING
PROJECT

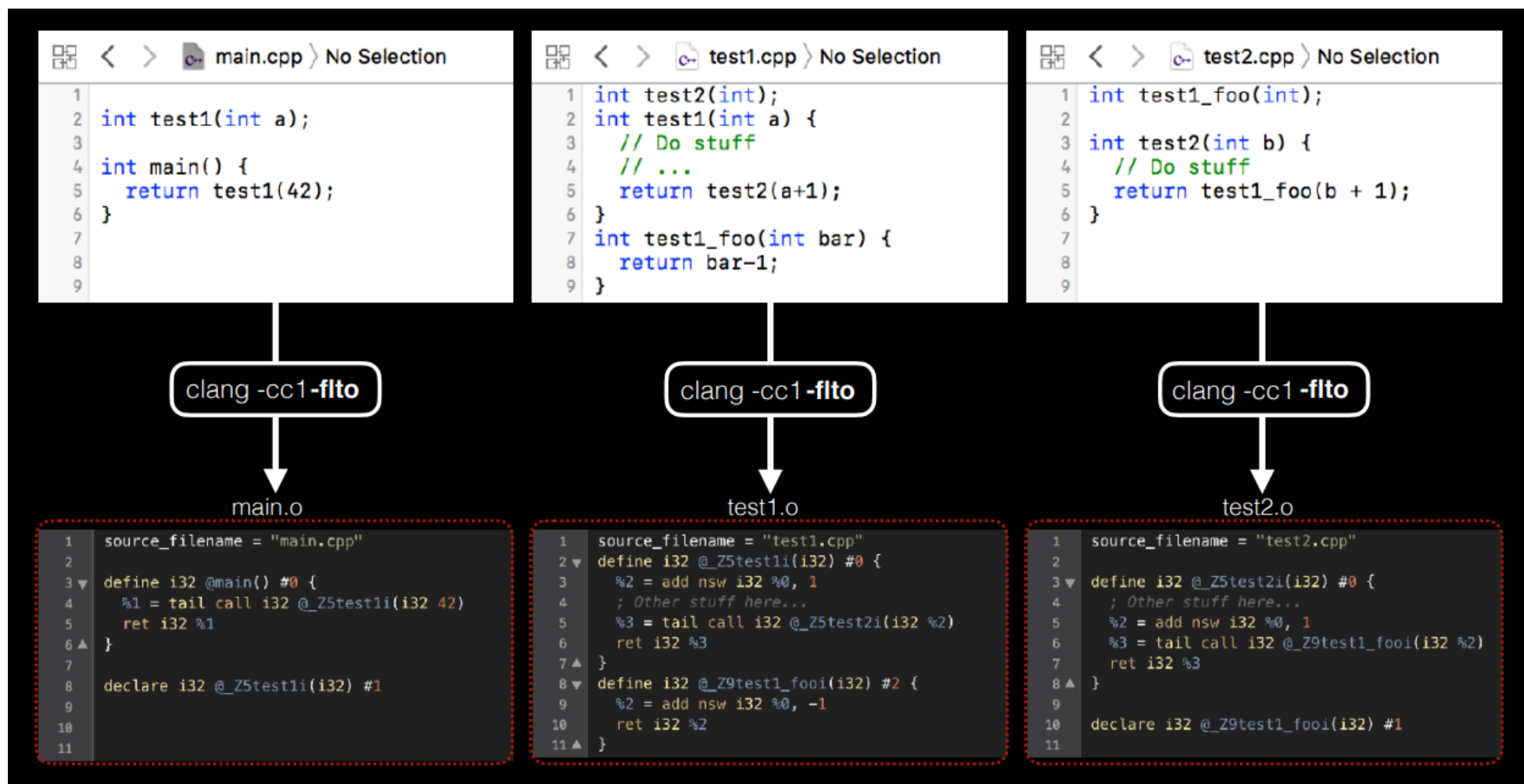
PGO

```
define void @f(i1 %a) {  
entry:  
    ...  
    br i1 %a, label %t, label %f, !prof !0  
  
t:  
    ...  
    br label %exit  
  
f:  
    ...  
    br label %exit  
  
exit:  
    ret void  
}  
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

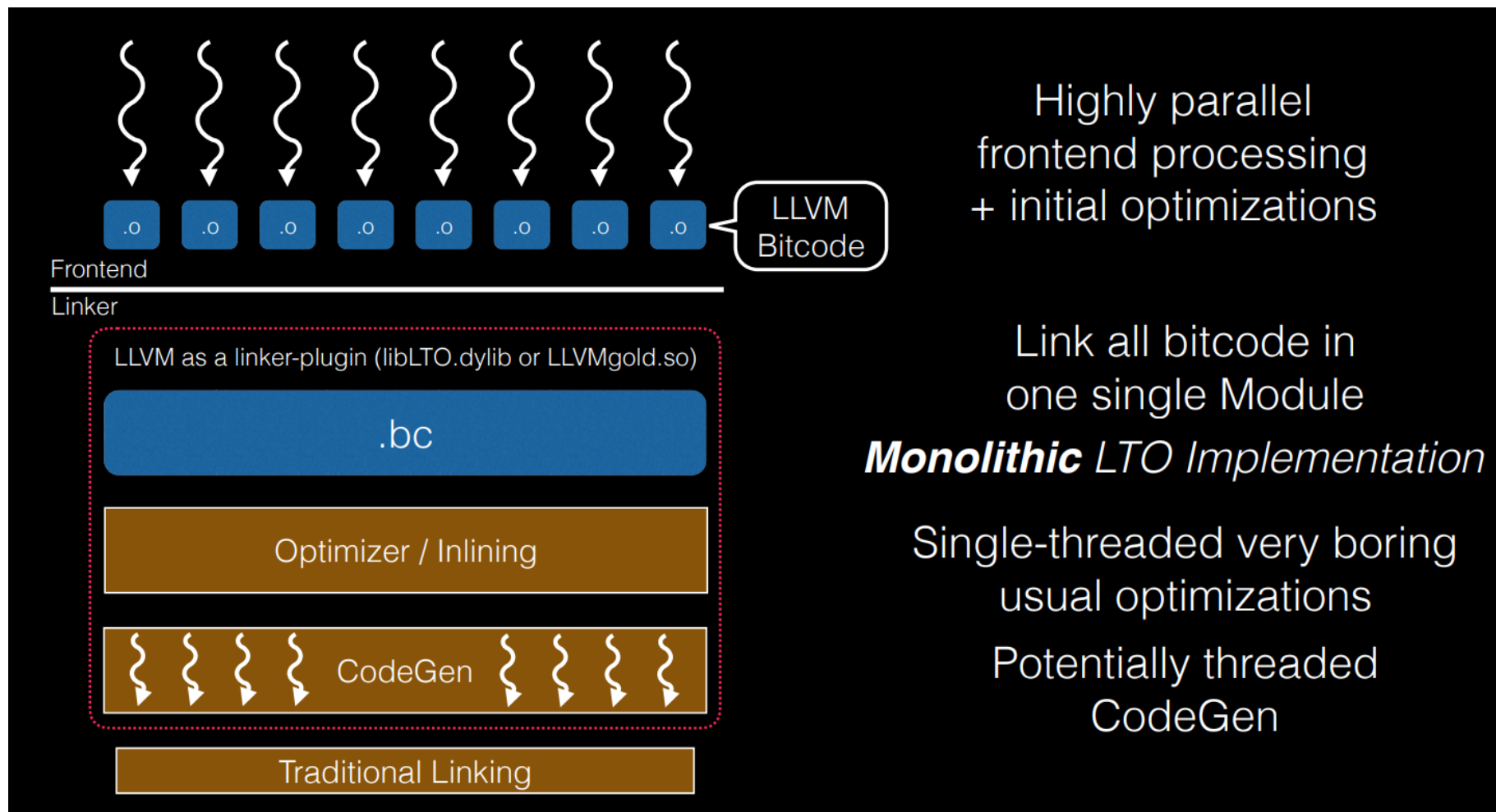
Link-Time Optimization



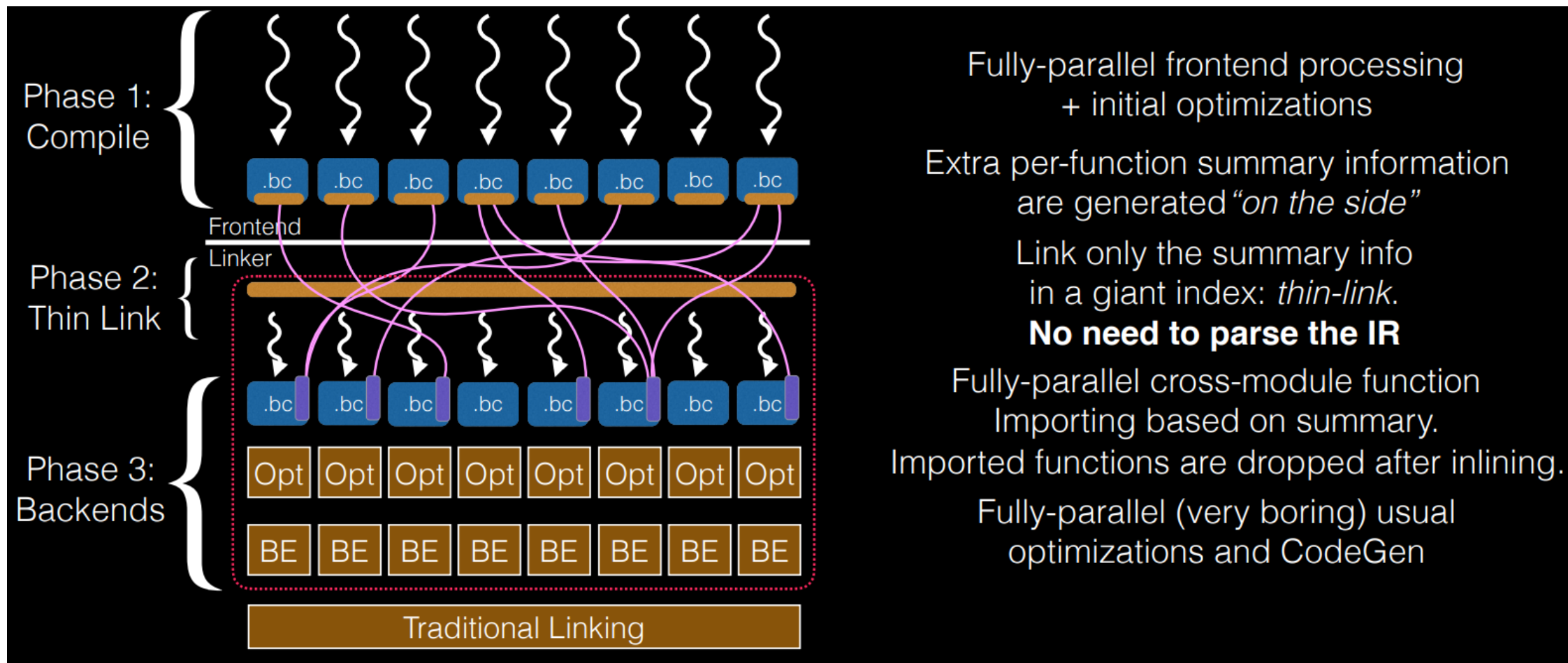
LTO



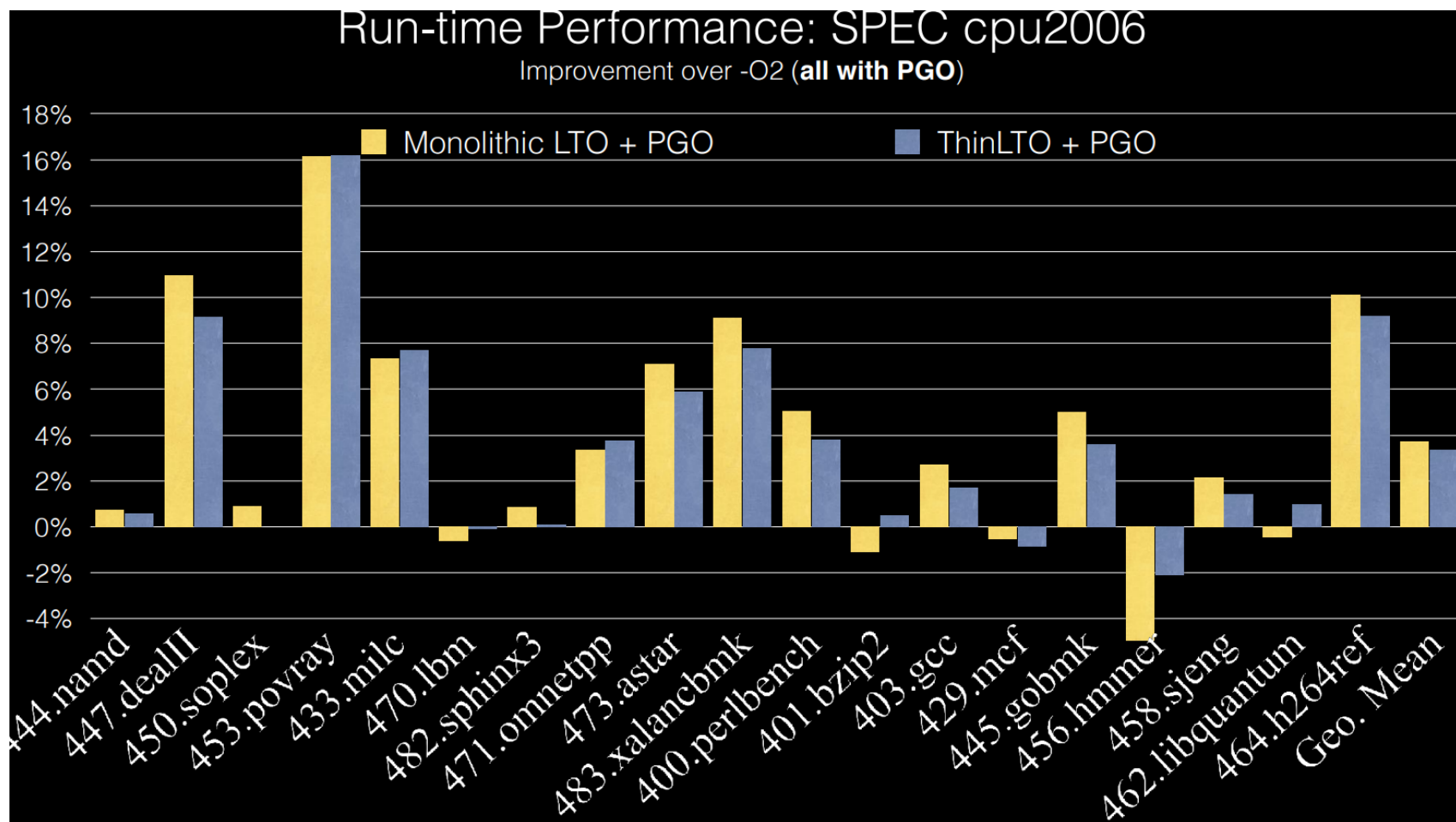
LTO



LTO



LTO

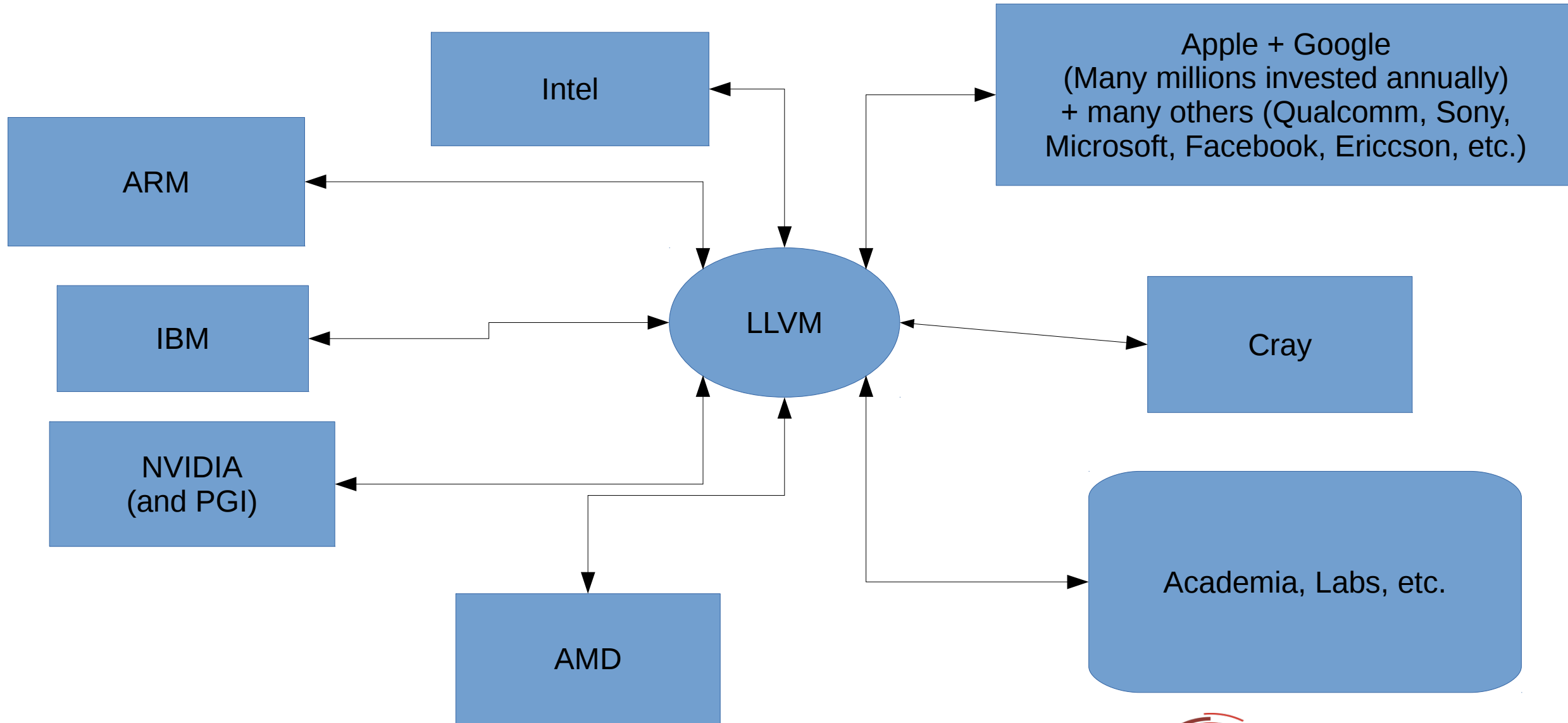


<http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf>

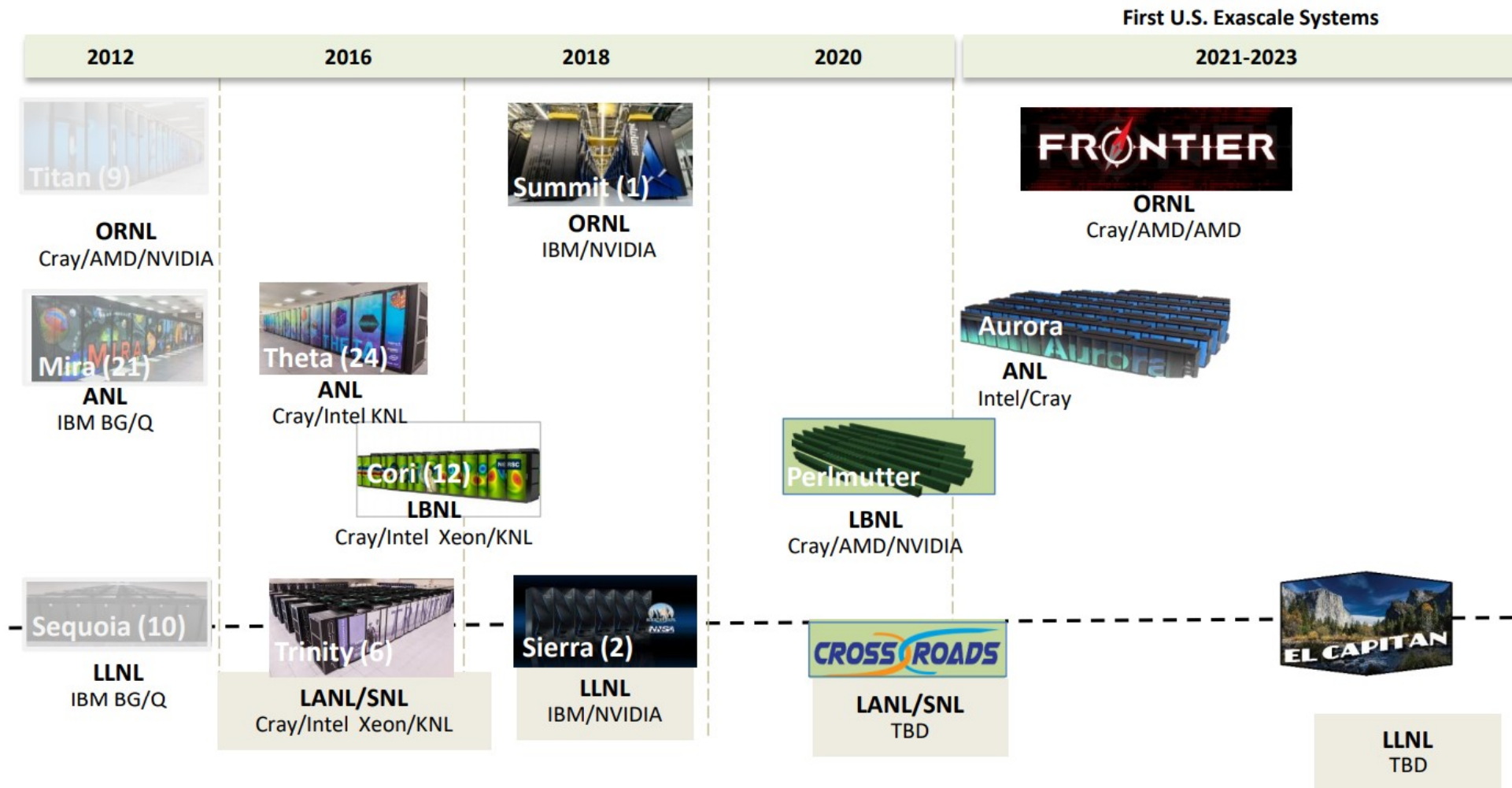


EXASCALE
COMPUTING
PROJECT

A role in exascale? Current/Future HPC vendors are already involved (plus many others)...



Pre-Exascale Systems [Aggregate Linpack (Rmax) = 323 PF!]



(https://science.osti.gov/-/media/ascr/ascac/pdf/meetings/201909/20190923_ASCAC-Helland-Barbara-Helland.pdf)

SOLLVE: OpenMP (WBS 2.3.1.13)

- Enhancing the implementation of OpenMP in LLVM:
 - Developing support for unified memory (e.g., from NVIDIA), kernel decomposition and pipelining, automated use of local memory, and other enhancements for accelerators.
 - Developing optimizations of OpenMP constructs to reduce overheads (e.g., from thread startup and barriers).
 - Building on LLVM parallel-IR work in collaboration with Intel.
- Using LLVM, Clang, and Flang to prototype new OpenMP features for standardization.
- Developing an OpenMP test suite, and as a result, testing and improving the quality of OpenMP in LLVM, Clang, and Flang.

PROTEAS: Parallel IR & More (WBS 2.3.2.09)

- Developing extensions to LLVM's intermediate representation (IR) to represent parallelism.
 - Strong collaboration with Intel and several academic groups.
 - Parallel IR can target OpenMP's runtime library among others.
 - Parallel IR can be targeted by OpenMP, OpenACC, and other programming models in Clang, Flang, and other frontends.
 - Building optimizations on parallel IR to reduce overheads (e.g., merging parallel regions and removing redundant barriers).
- Developing support for OpenACC in Clang, prototyping non-volatile memory features, and integration with Tau performance tools.

Y-Tune: Autotuning (WBS 2.3.2.07)

- Enhancing LLVM to better interface with autotuning tools.
- Enhancing LLVM's polyhedral loop optimizations and the ability to drive them using autotuning.
- Using Clang, and potentially Flang, for parsing and semantic analysis.

Kitsune: LANL ATDM Dev. Tools (WBS 2.3.2.02)

- Using parallel IR to replace template expansion in FleCSI, Kokkos, RAJA, etc.
- Enhanced parallel-IR optimizations and targeting of various runtimes/architectures.
- Flang evaluation, testing, and Legion integration, plus other programming-model enhancements.
- ByFI: Instrumentation-based performance counters using LLVM.

Flang: LLVM Fortran Frontend (WBS 2.3.5.06)

- Working with NVIDIA (PGI), ARM, and others to develop an open-source, production-quality LLVM Fortran frontend.
 - Can target parallel IR to support OpenMP (including OpenMP offloading) and OpenACC.

Note: The proxy-apps project (WBS 2.2.6.01) is also enhancing LLVM's test suite.

Composition of Transformations

Order is Important

```
#pragma omp unroll factor(2)
#pragma omp reverse
for (int i = 0; i < 128; i+=1)
    Stmt(i);
```



```
#pragma omp unroll factor(2)
for (int i = 127; i >= 0; i-=1)
    Stmt(i);
```



```
for (int i = 127; i >= 0; i-=1) {
    Stmt(i);
    Stmt(i-1);
}
```

```
#pragma omp reverse
#pragma omp unroll factor(2)
for (int i = 0; i < 128; i+=1)
    Stmt(i);
```



```
#pragma omp reverse
for (int i = 0; i < 128; i+=2) {
    Stmt(i);
    Stmt(i+1);
}
```



```
for (int i = 126; i >= 0; i-=2) {
    Stmt(i);
    Stmt(i+1);
}
```

Matrix-Matrix Multiplication

```
void matmul(int M, int N, int K,  
            double C[const restrict static M][N],  
            double A[const restrict static M][K],  
            double B[const restrict static K][N]) {  
    #pragma clang loop(j2) pack array(A)  
    #pragma clang loop(i1) pack array(B)  
    #pragma clang loop(i1,j1,k1,i2,j2) interchange \  
                                permutation(j1,k1,i1,j2,i2)  
    #pragma clang loop(i,j,k) tile sizes(96,2048,256) \  
                                pit_ids(i1,j1,k1) tile_ids(i2,j2,k2)  
    #pragma clang loop id(i)  
    for (int i = 0; i < M; i += 1)  
        #pragma clang loop id(j)  
        for (int j = 0; j < N; j += 1)  
            #pragma clang loop id(k)  
            for (int k = 0; k < K; k += 1)  
                C[i][j] += A[i][k] * B[k][j];  
}
```


Matrix-Matrix Multiplication

After Transformation

```
double Packed_B[256][2048];
double Packed_A[96][256];
if (runtime check) {
    if (M >= 1)
        for (int c0 = 0; c0 <= floord(N - 1, 2048); c0 += 1)    // Loop j1
            for (int c1 = 0; c1 <= floord(K - 1, 256); c1 += 1) { // Loop k1

                // Copy-in: B -> Packed_B
                for (int c4 = 0; c4 <= min(2047, N - 2048 * c0 - 1); c4 += 1)
                    for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1)
                        Packed_B[c4][c5] = B[256 * c1 + c5][2048 * c0 + c4];

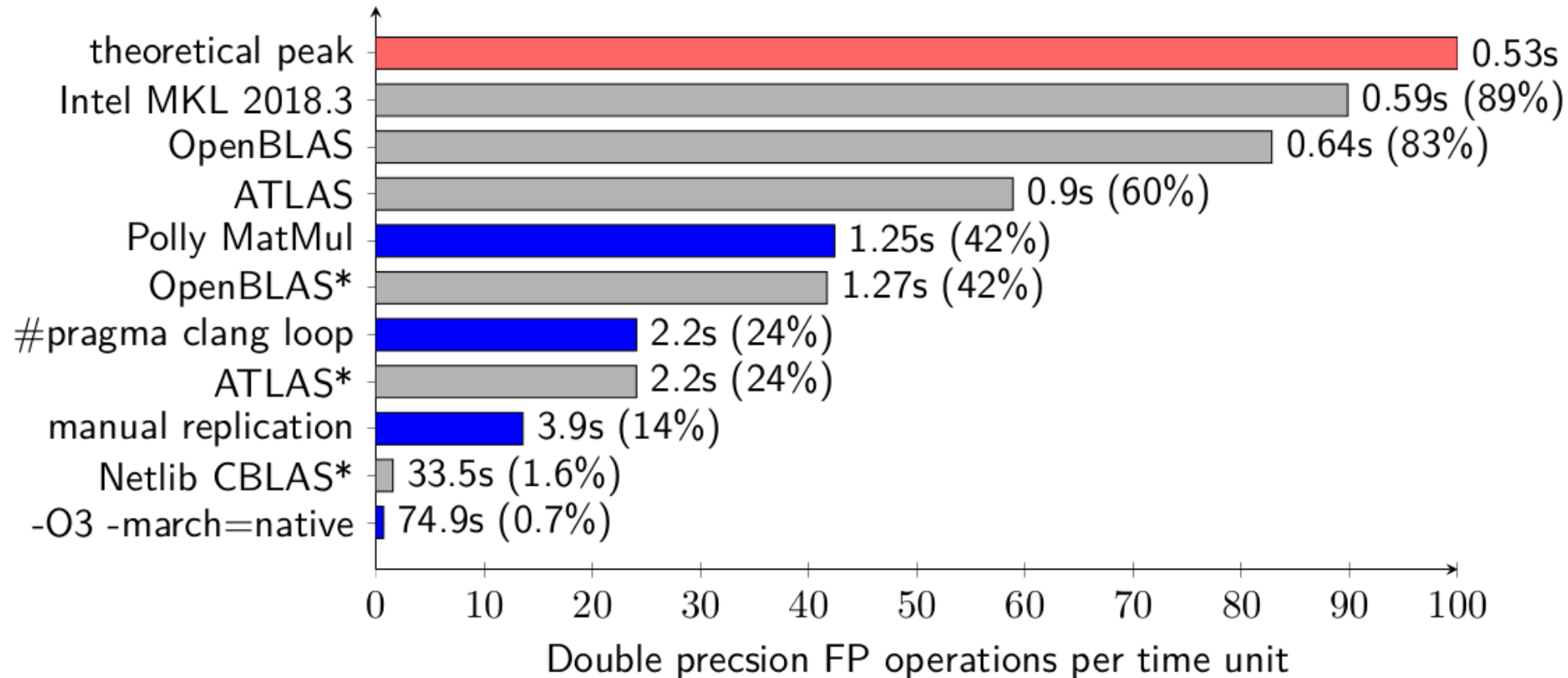
                for (int c2 = 0; c2 <= floord(M - 1, 96); c2 += 1) { // Loop i1

                    // Copy-in: A -> Packed_A
                    for (int c6 = 0; c6 <= min(95, M - 96 * c2 - 1); c6 += 1)
                        for (int c7 = 0; c7 <= min(255, K - 256 * c1 - 1); c7 += 1)
                            Packed_A[c6][c7] = A[96 * c2 + c6][256 * c1 + c7];

                    for (int c3 = 0; c3 <= min(2047, N - 2048 * c0 - 1); c3 += 1)    // Loop j2
                        for (int c4 = 0; c4 <= min(95, M - 96 * c2 - 1); c4 += 1)    // Loop i2
                            for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1) // Loop k2
                                C[96 * c2 + c4][2048 * c0 + c3] += Packed_A[c4][c5] * Packed_B[c3][c5];
                }
            }
} else {
    /* original code */
}
```

Matrix-Matrix Multiplication

Execution Speed



* Pre-compiled from Ubuntu repository

What To Do With OpenACC Code?

Clacc: OpenACC Support for Clang and LLVM

Who

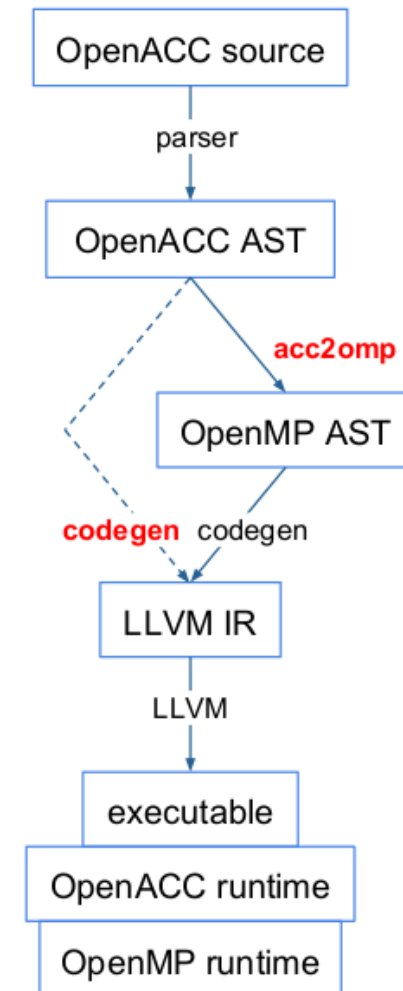
- Joel E. Denny (ORNL)
- Seyong Lee (ORNL)
- Jeffrey S. Vetter (ORNL)

Where

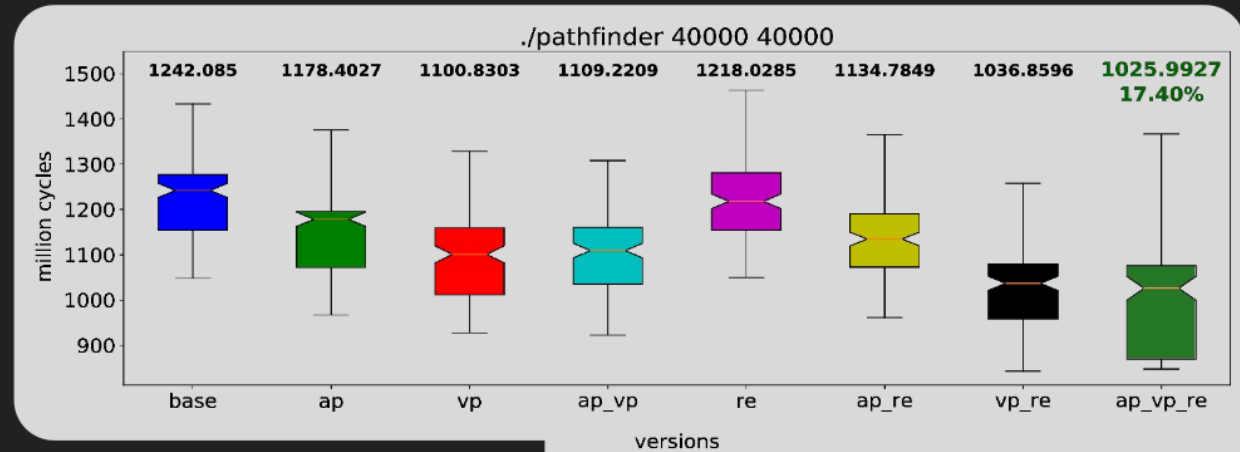
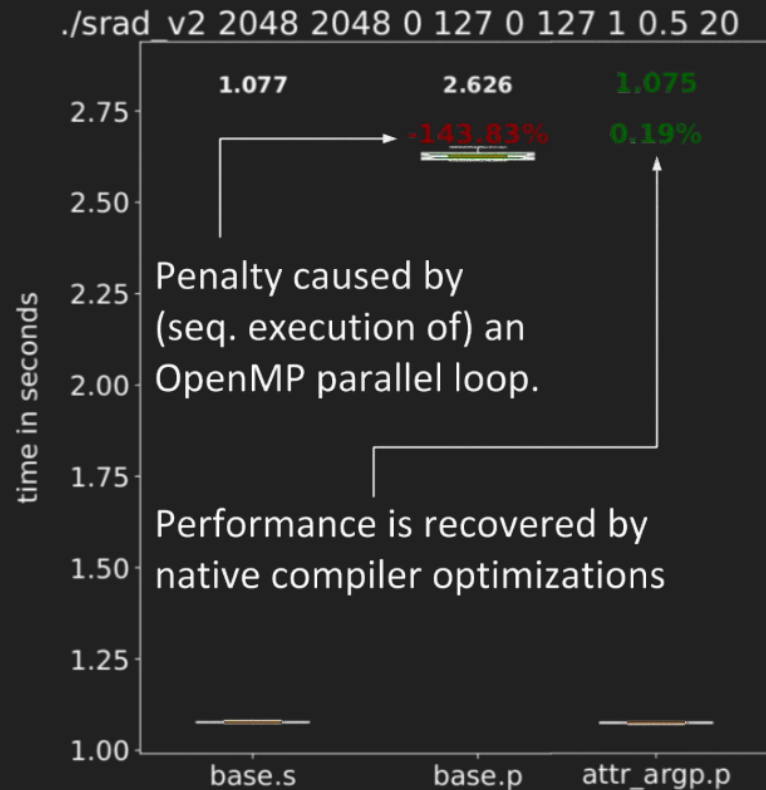
- Clacc: Translating OpenACC to OpenMP in Clang, Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Dallas, TX, USA, (2018).
- <https://ft.ornl.gov/research/clacc>
- Clacc Poster (Wed at ECP AHM)

What

- Develop production-quality, standard-conforming traditional OpenACC compiler and runtime support by extending Clang and LLVM
- Enable research and development of source-level OpenACC tools
 - Design compiler to leverage Clang/LLVM ecosystem extensibility
 - E.g., Pretty printers, analyzers, lint tools, and debugger and editor extensions
- As matures, contribute OpenACC support to upstream Clang and LLVM
- Throughout development
 - Actively contribute upstream all mutually beneficial Clang and LLVM improvements
 - Actively contribute to the OpenACC specification



Optimizing Parallel Programs with LLVM



Reuse “scalar” code

- constant propagation
- argument promotion
- attribute deduction
- ...

New “parallel” optimizations

- barrier elimination
- parallel region expansion
- parallelism aware code motion
- ...

See our **IWOMP'18** & **LCPC'18** papers, as well as the **LLVMDev'18** talk/video!

Opportunities for the Future

- Race-Detection Tools and other Sanitizers in HPC
 - Scalable Data Collection
 - Integration with MPI or other inter-node communication frameworks
 - Support on GPUs and other accelerators
- More static analysis, both frontend and optimizer, for HPC
 - Support for MPI
 - Support for Fortran
 - Support for GPUs and other accelerators
 - Support for advanced loop optimizations and other user-directed optimizations
- FDR-like capabilities for large-scale HPC applications
 - Debugging crashes at scale is hard.
- Integrated dynamic and static performance analysis (e.g., using MCA-like capabilities)
 - Better understanding of performance counters
 - Understanding of working sets and cache populations
 - Support for GPUs and other accelerators
- Better support for LTO and PGO in HPC environments
 - Scalable data collection (for PGO)
 - Build-system integration, LTO-enabled libraries, etc.
 - Support for GPUs and other accelerators

Acknowledgments

Thanks to ALCF, ANL, ECP, DOE, and the LLVM community!

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

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.