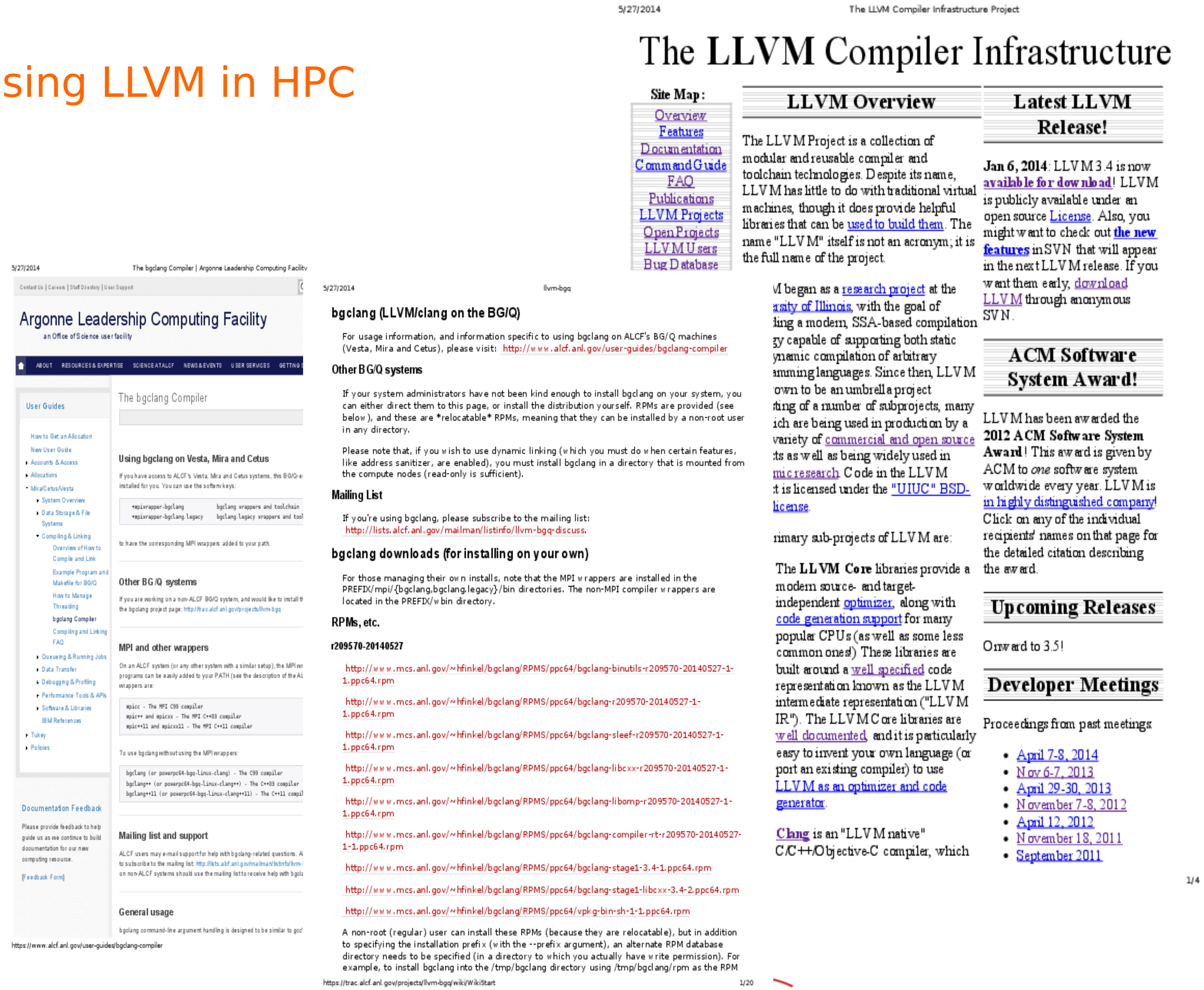
The Many Faces of Instrumentation:

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

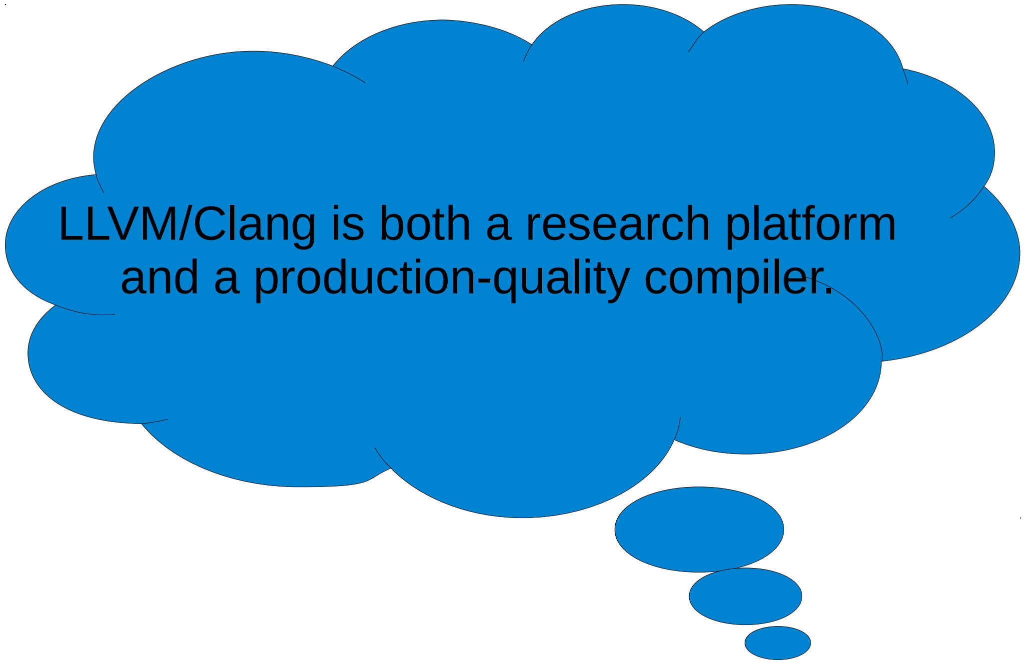
Hal Finkel
Leadership Computing Facility
Argonne National Laboratory

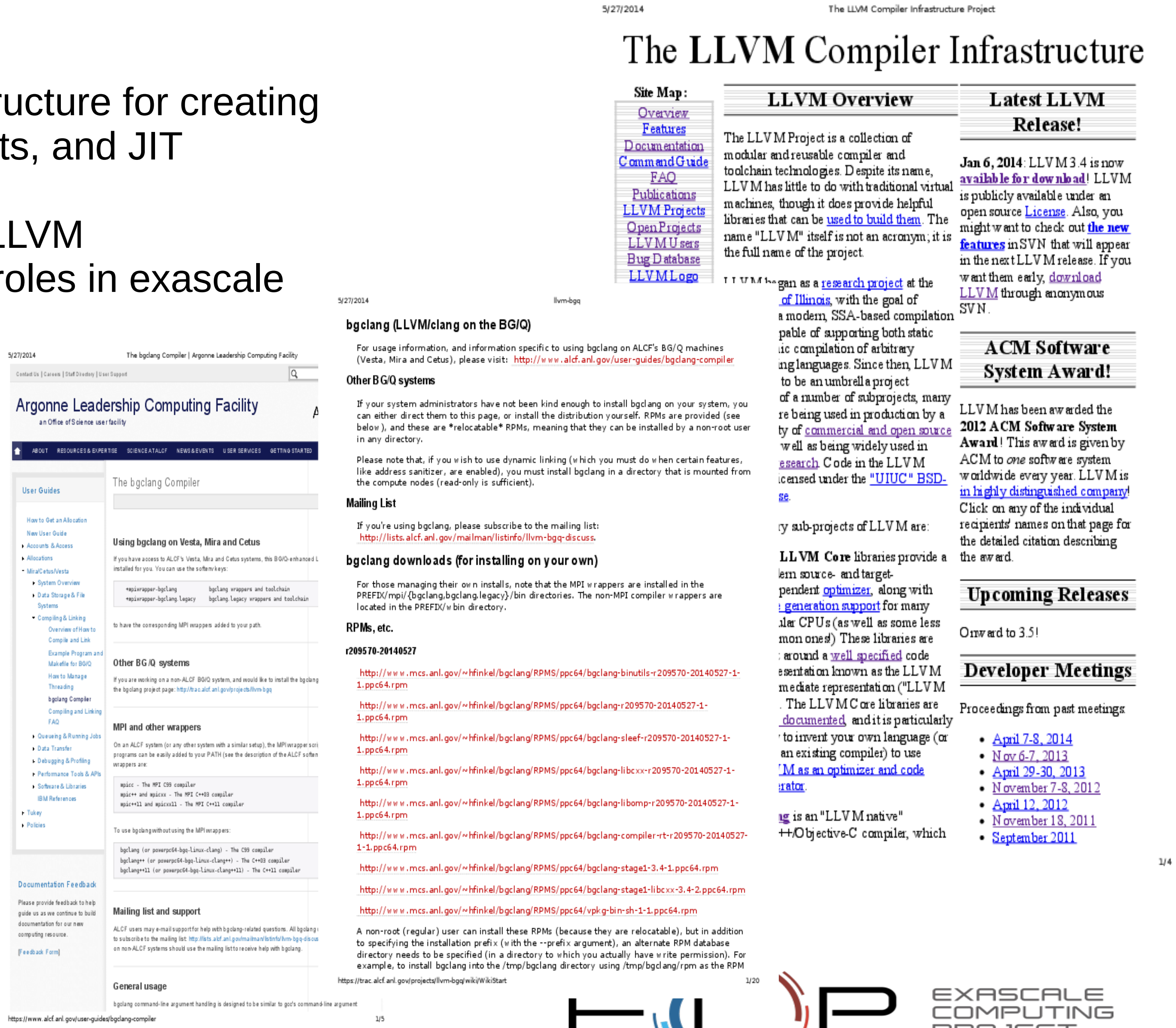
hfinkel@anl.gov

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

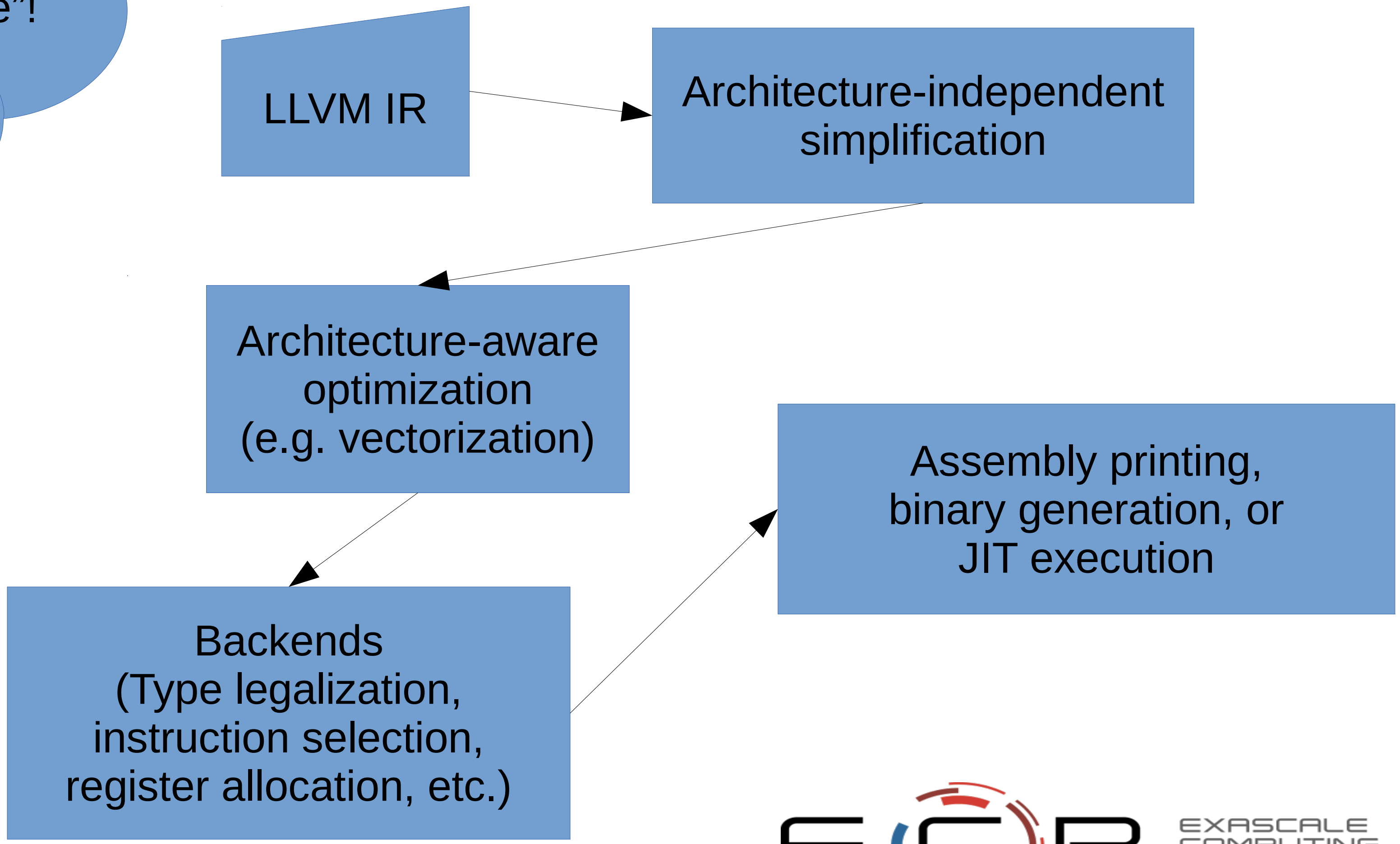




What is LLVM:

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

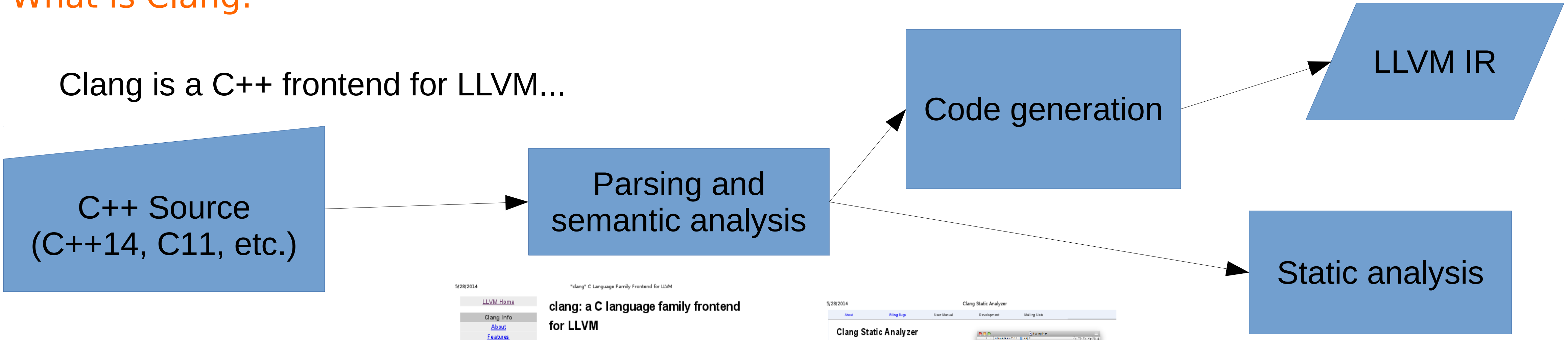
LLVM is a multi-architecture infrastructure for constructing compilers and other toolchain components.



EXASCALE COMPUTING PROJECT

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

LLVM Home

Clang Info
About
Features
Comparisons
Related Projects
User's Manual
Language Compatibility
Language Extensions
C++ Status

Clang Development
Get Started
Get Involved
Open Projects
Clang Internals
Hacking on Clang

Clang Tools
Automatic Bug-Finding
Writing Clang Tools

Communication
cfe-users List
cfe-dev List
cfe-commits List
Bug Reports
Planet Clang
IRC: irc.oftc.net#llvm

The Code
Check Out SVN
Browse SVN
Browse ViewVC
doxygen

Quick Links
Testing Coverage

Clang Events
October 2009
November 2010
LLVM events

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++ front-end 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 compiles 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:

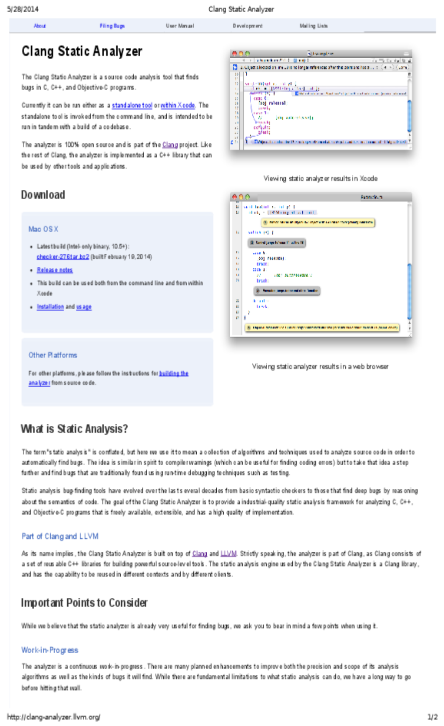
- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

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/ 1/2



5/28/2014 Clang Static Analyzer

About Filing Bugs User Manual Development Mailing Lists

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 Xcode. The standalone tool is invoked from the command line, and is intended to be run in tandem with a build of a codebase.

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.

Viewing static analyzer results in Xcode

Download

Mac OS X

- Latest build (Intel-only binary, 10.5+): checker-276.tar.bz2 (built February 19, 2014)
- 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.

Viewing static analyzer results in a web browser

What is Static Analysis?

The term "static analysis" is conflated, 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 in spirit to compiler warnings (which can be useful for finding coding errors) but to take that idea a step further and find bugs that are traditionally found using run-time 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 industrial-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 source-level tools. The static analysis engine used by the Clang Static Analyzer is a Clang library, and has the capability to be reused in different contexts and by different clients.

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/ 1/2

EXASCALE COMPUTING PROJECT

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 compili Objective-C code in a debug configuration), extremely useful error and warning messages and to provide a platform for building great source level t . 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 g 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 expr , LLVM JIT, LLVM disassembler, etc so that it provides an experience that "just works". It is also blazing fast and much more memory GDB at loading symbols.
4. The libc++ and libc++ ABI projects provide a standard conformant and high-performance implementation of the C++ Standa g full support for C++11.
5. The compiler-rt project provides highly tuned implementations of the low-level code generator support routines like r calls generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It a ns of run-time libraries for dynamic testing tools such as AddressSanitizer, ThreadSanitizer, MemorySanitizer, and DataFlow
6. The OpenMP subproject provides an OpenMP runtime for use with the OpenMP implementation in Clan
7. The polly project implements a suite of cache-locality optimizations as well as auto-parallelism and del.
8. The libclc project aims to implement the OpenCL standard library.

valgrind.

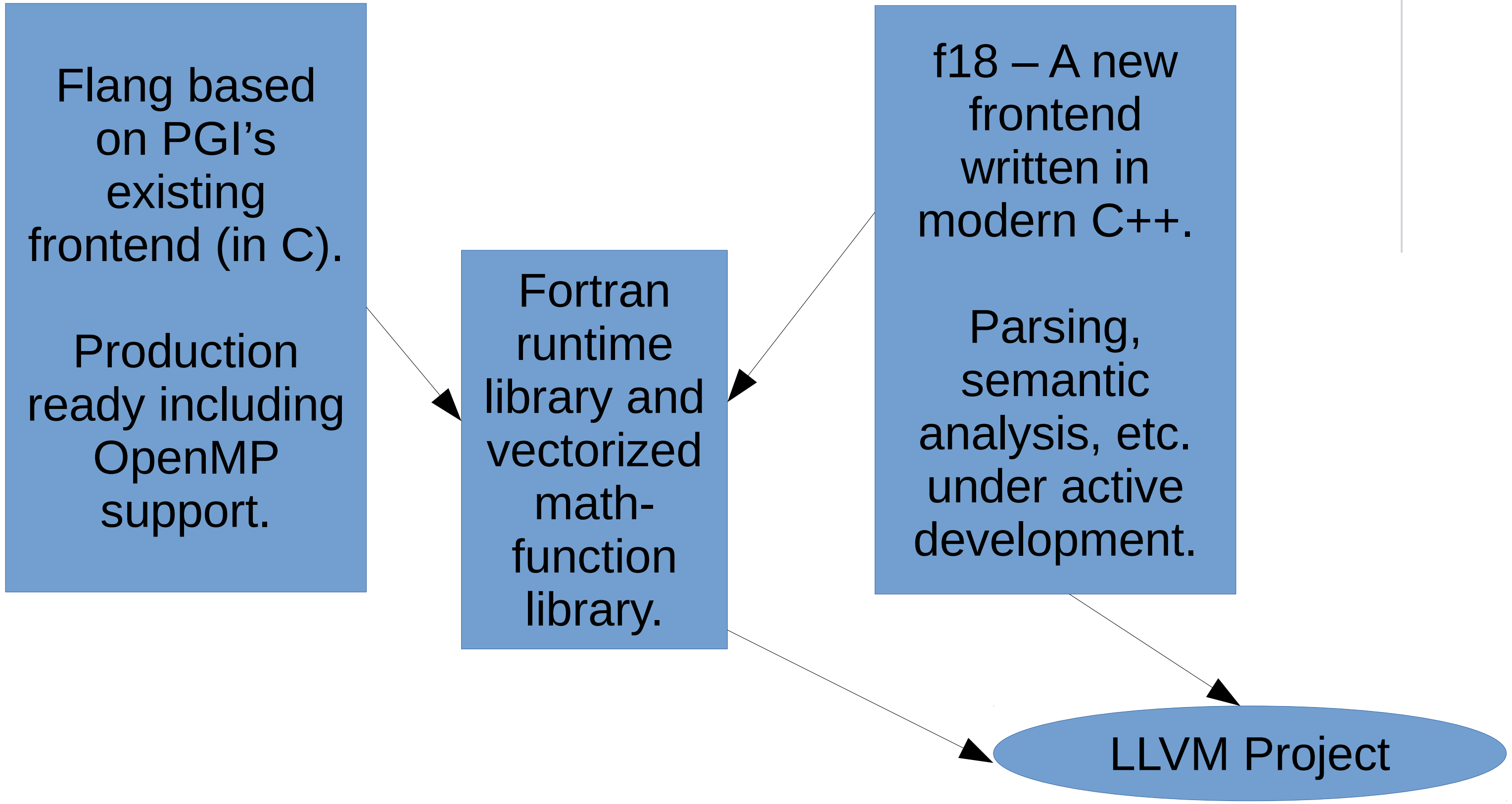
11. The lld project aims to be the built-in linker for clang/llvm. Currently, clang must invoke the system linker to produce executables.

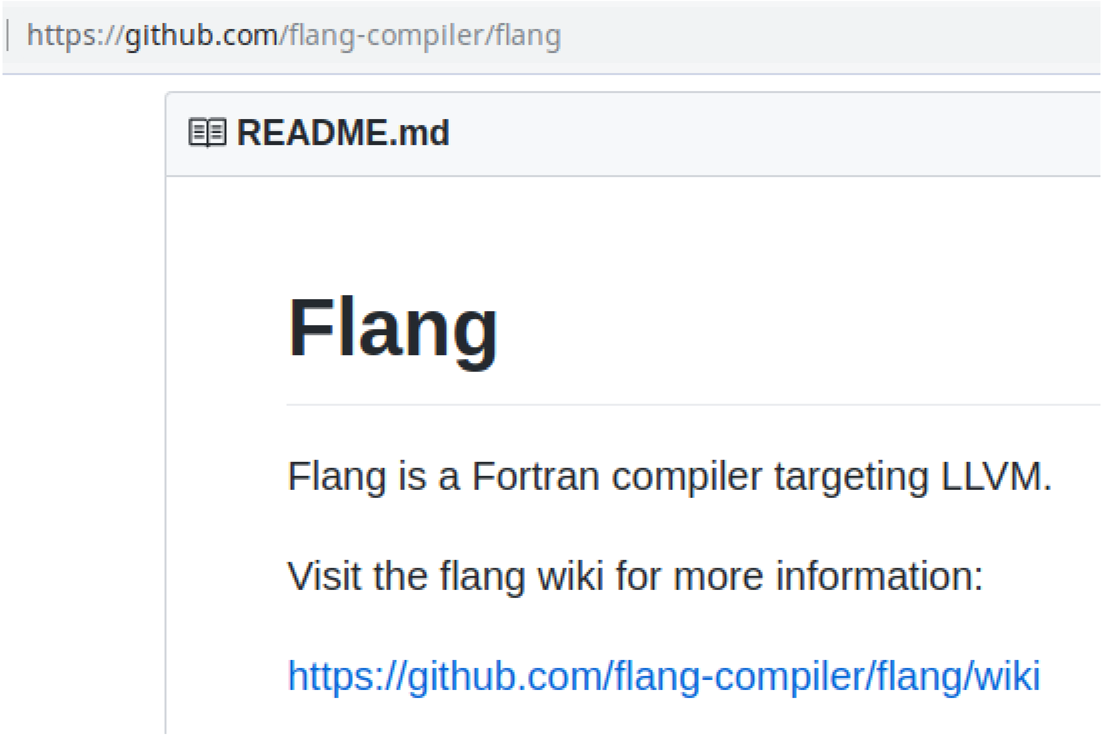
The core LLVM compiler-infrastructure components are one of the subprojects in the LLVM project. These components are also referred to as "LLVM."

EXASCALE COMPUTING PROJECT

What About Flang?

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




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

README.md

Flang

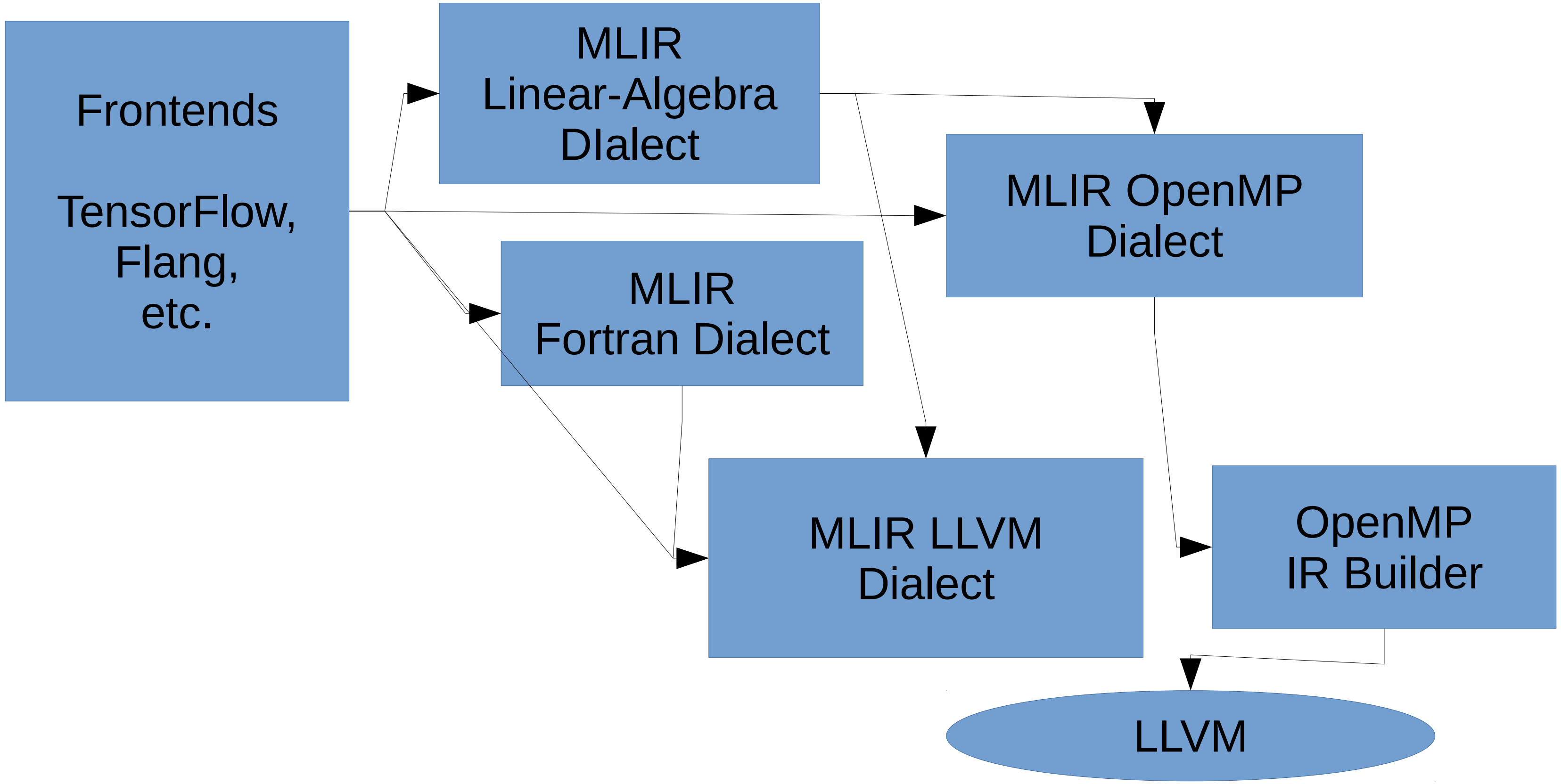
Flang is a Fortran compiler targeting LLVM.

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

EXASCALE COMPUTING PROJECT

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.

EXASCALE COMPUTING PROJECT

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
```

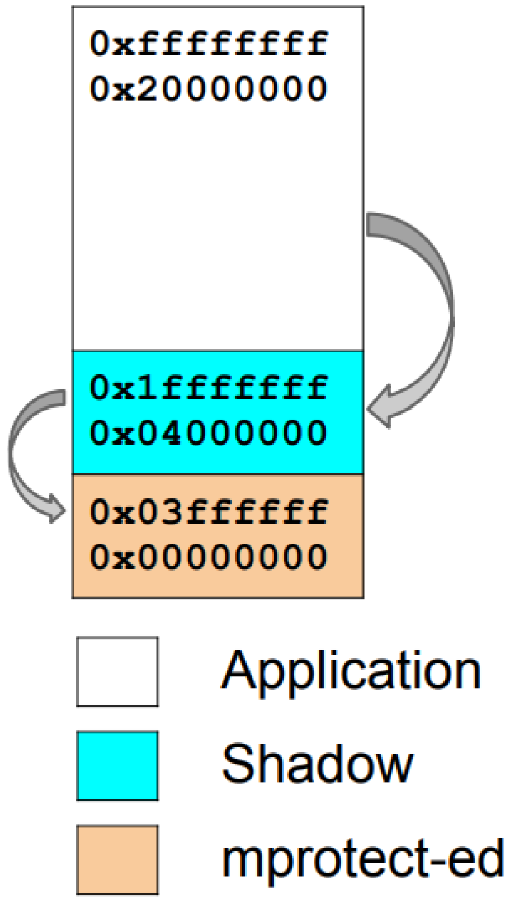
http://www.llvm.org/devmtg/2012-11/Serebryany_TSan-MSan.pdf

EXASCALE COMPUTING PROJECT

Address Sanitizer

ASan shadow memory

Virtual address space



Instrumentation

```
*a = ...
```

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

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

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_stru
ct> >, 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_stru
ct> >, 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
```


EXASCALE COMPUTING PROJECT

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.

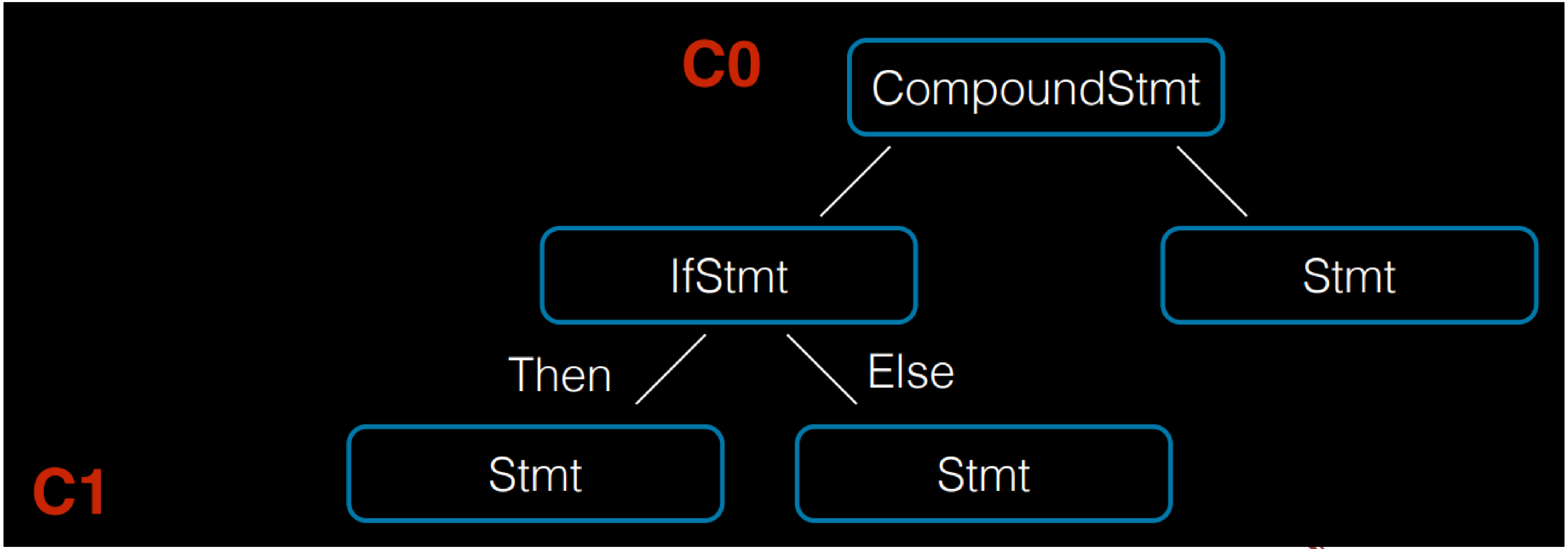
```
Cycles with backend pressure increase [ 48.07% ]
Throughput Bottlenecks:
  Resource Pressure       [ 47.77% ]
  - JFPA  [ 47.77% ]
  - JFPU0  [ 47.77% ]
  Data Dependencies:      [ 0.30% ]
  - Register Dependencies [ 0.30% ]
  - Memory Dependencies   [ 0.00% ]

Critical sequence based on the simulation:

              Instruction                         Dependency Information
 +----< 2.    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

EXASCALE COMPUTING PROJECT

Profile-Guided Optimization

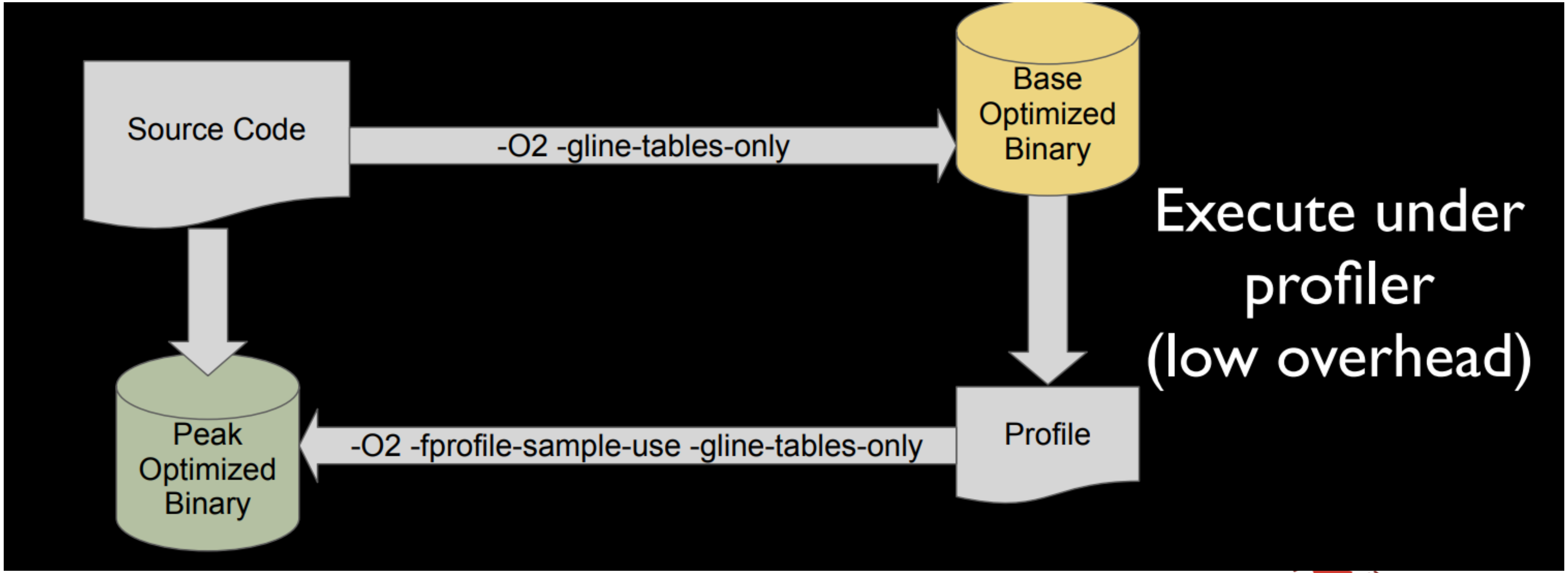
Instrumentation vs. Sampling PGO; for instrumentation:



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

PGO

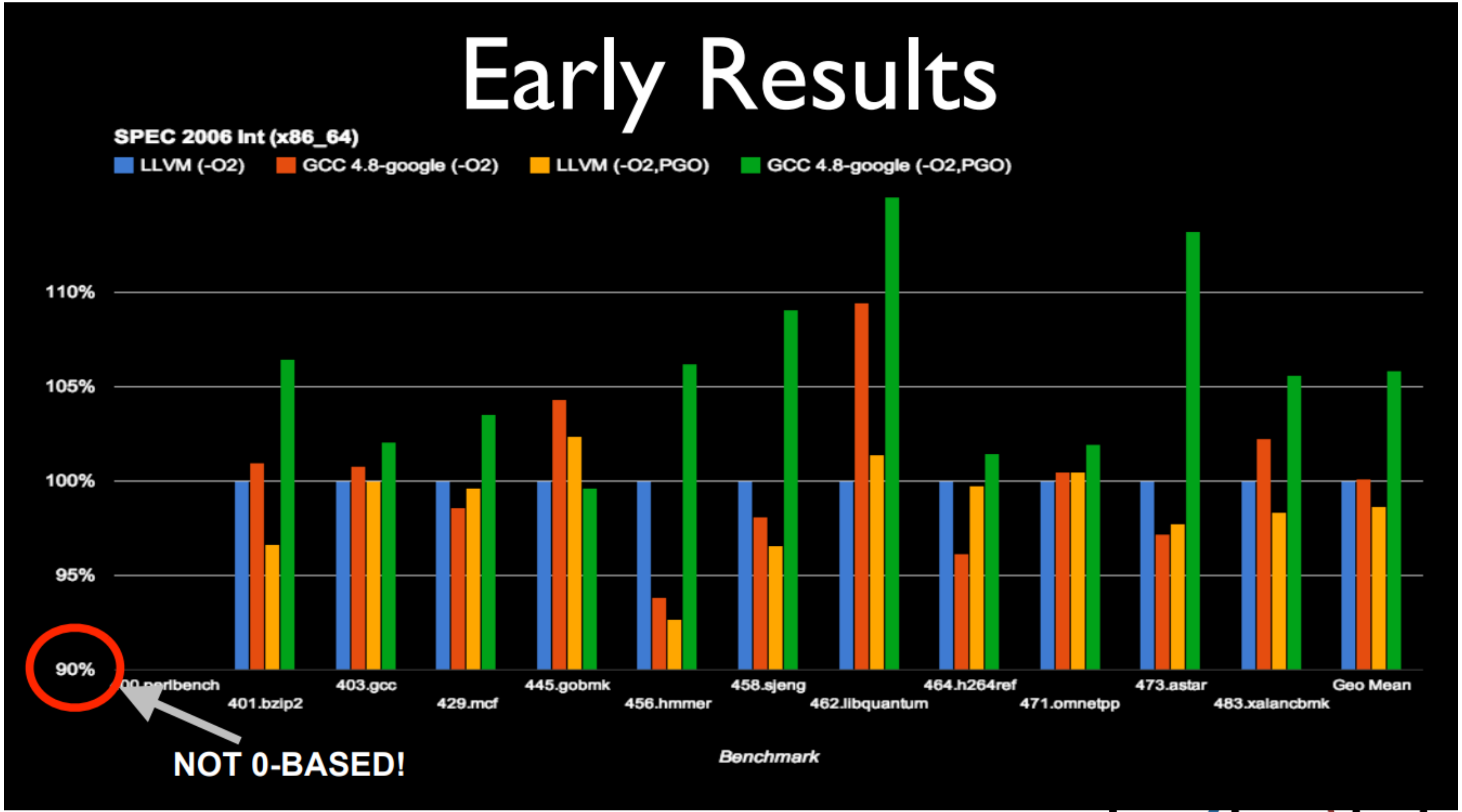
Instrumentation vs. Sampling PGO; for sampling:



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

EXASCALE COMPUTING PROJECT

PGO



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

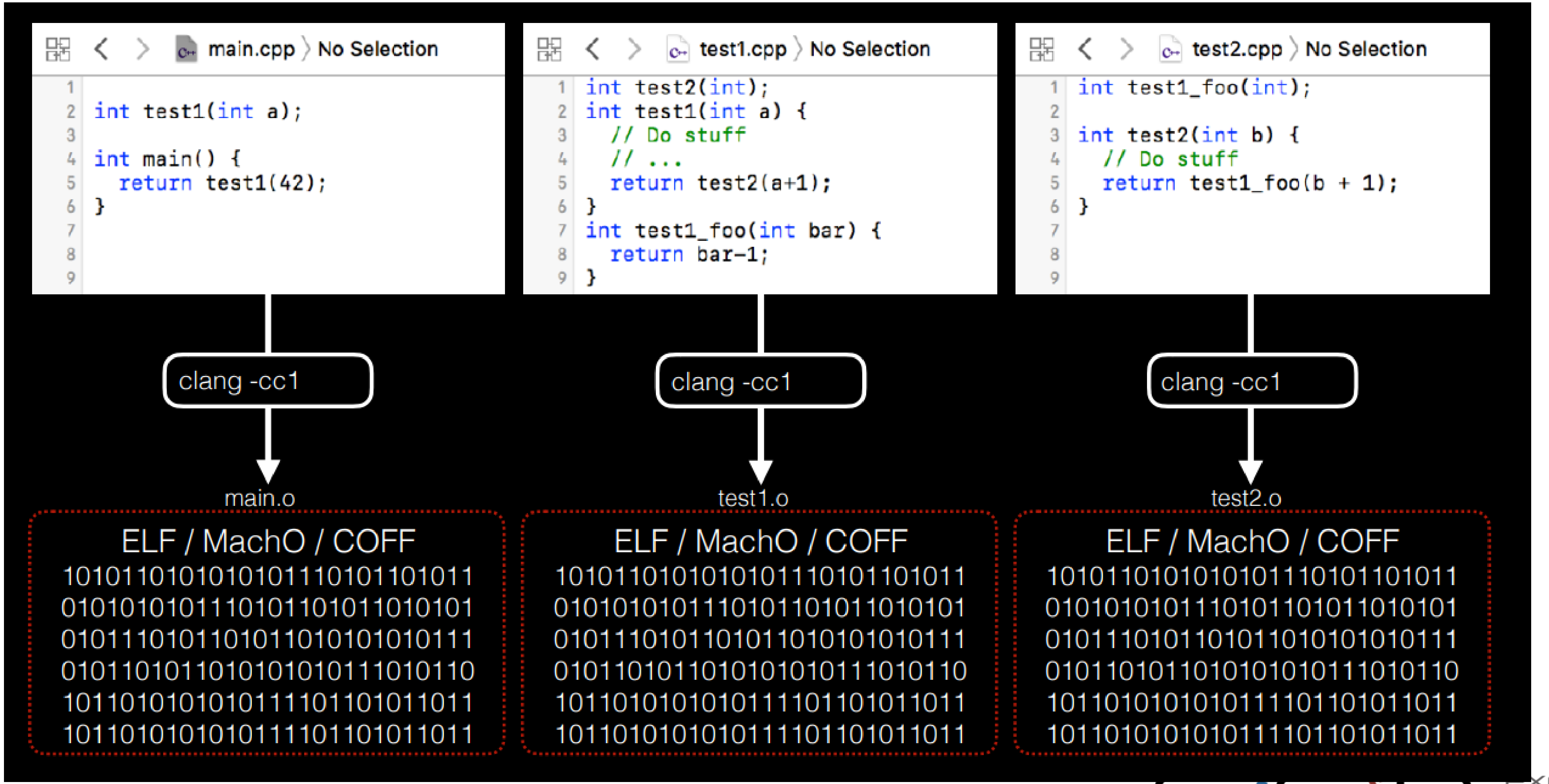
EXASCALE COMPUTING PROJECT

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}
```

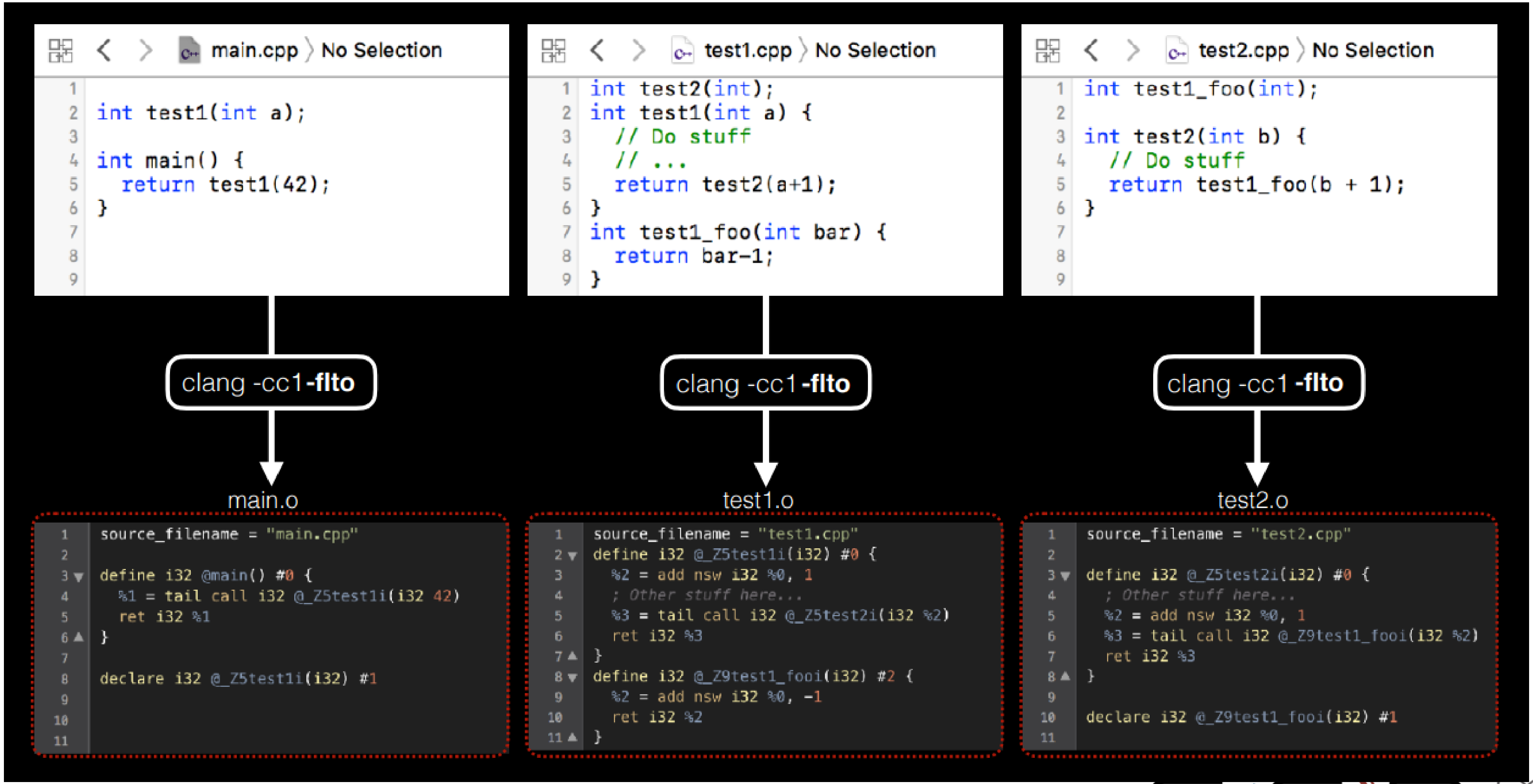
https://llvm.org/devmtg/2013-11/slides/Carruth-PGO.pdf

Link-Time Optimization



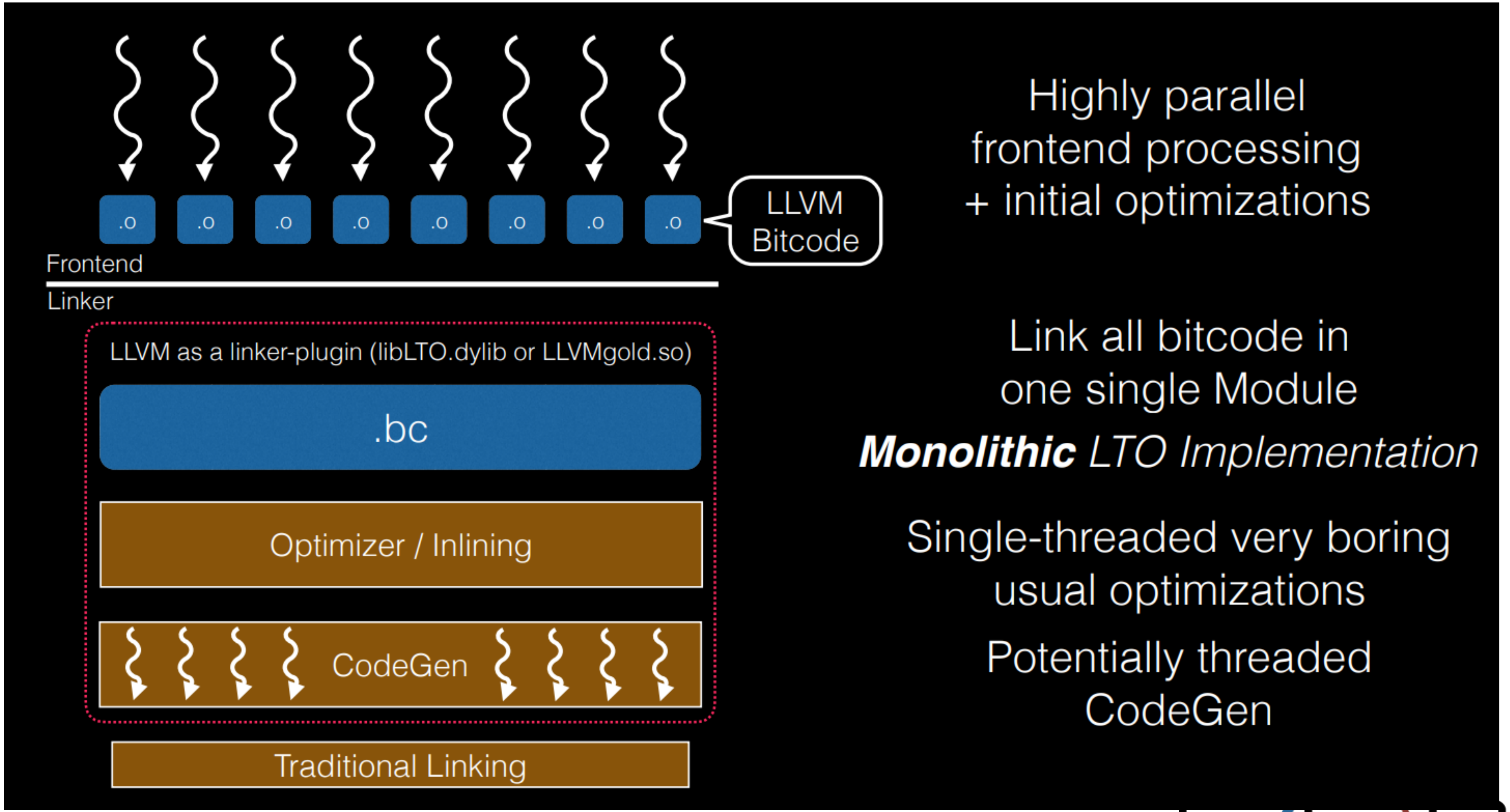
http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf

LTO



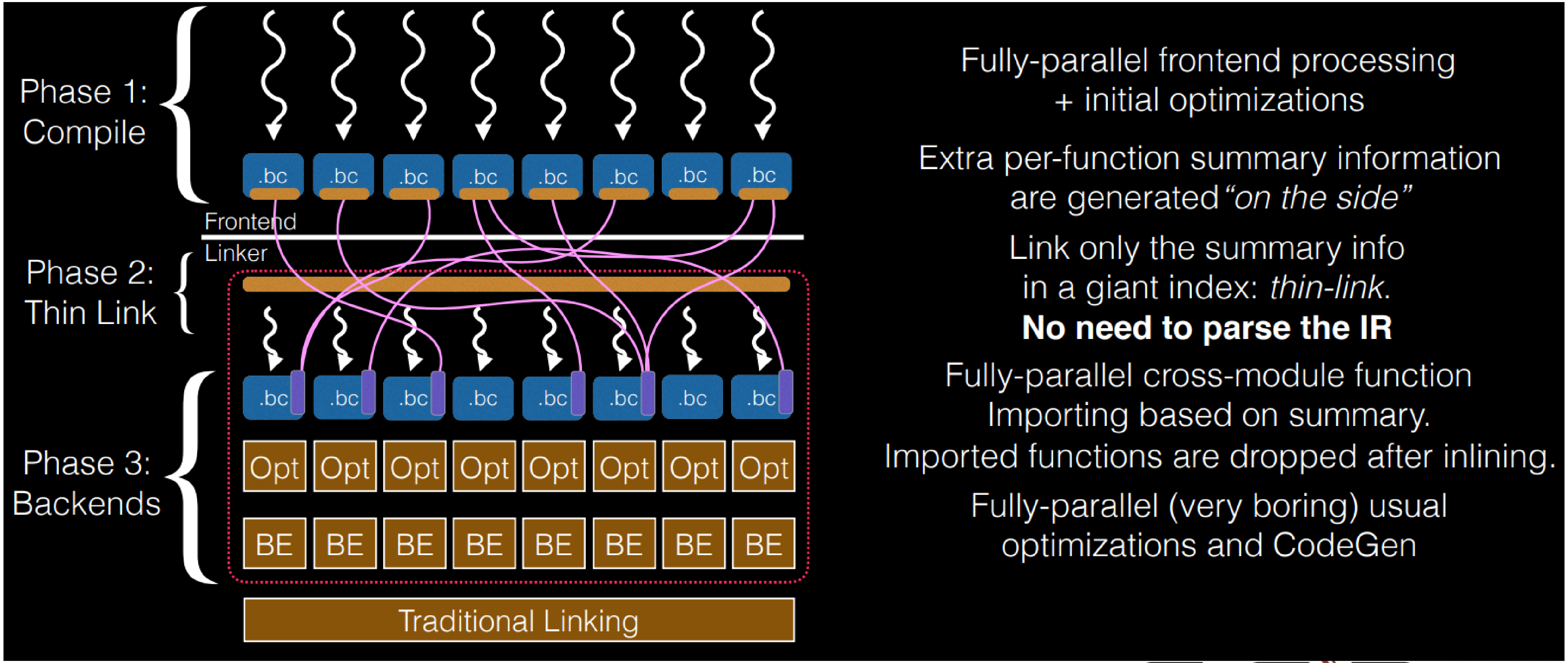
http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf

LTO



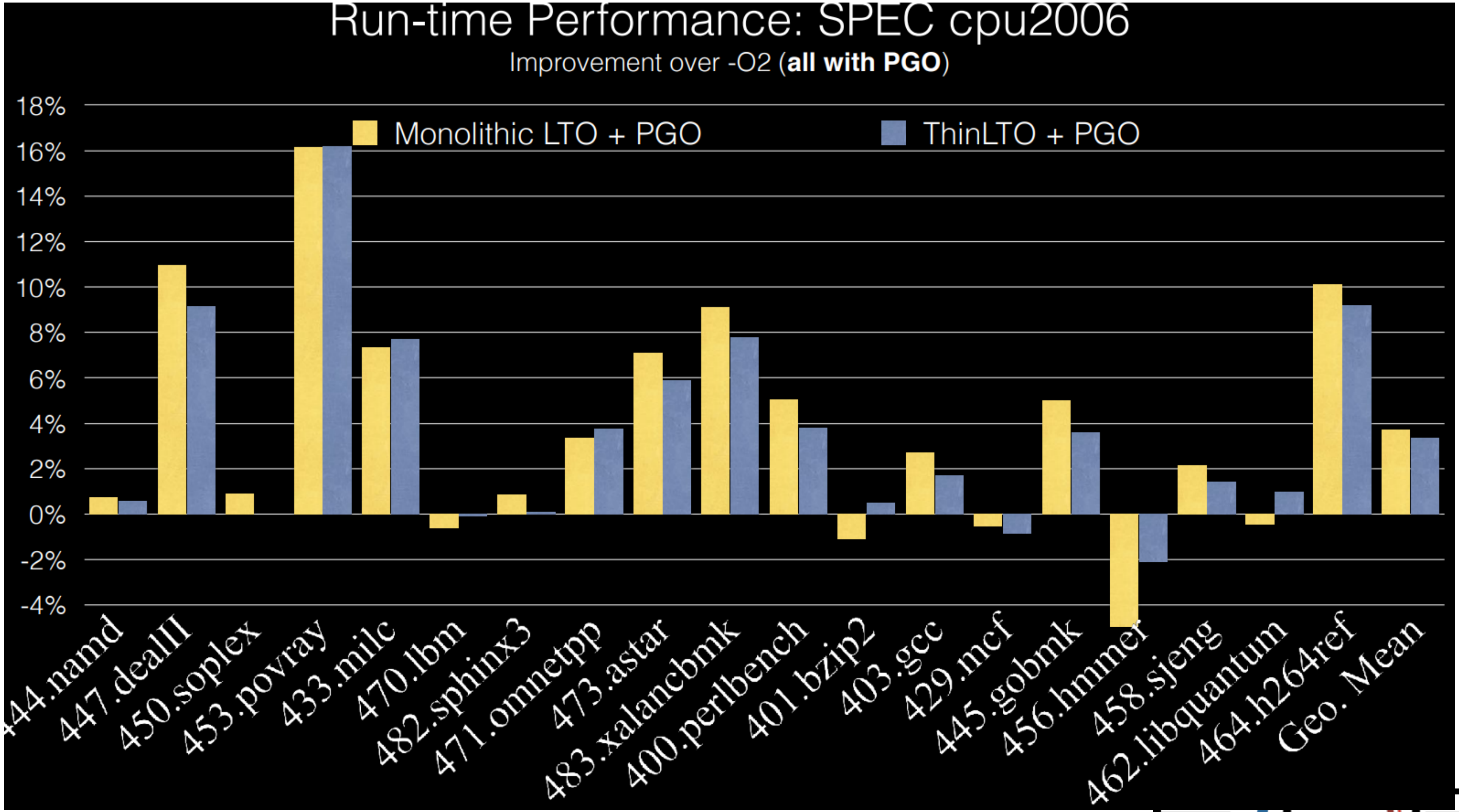
http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf

LTO



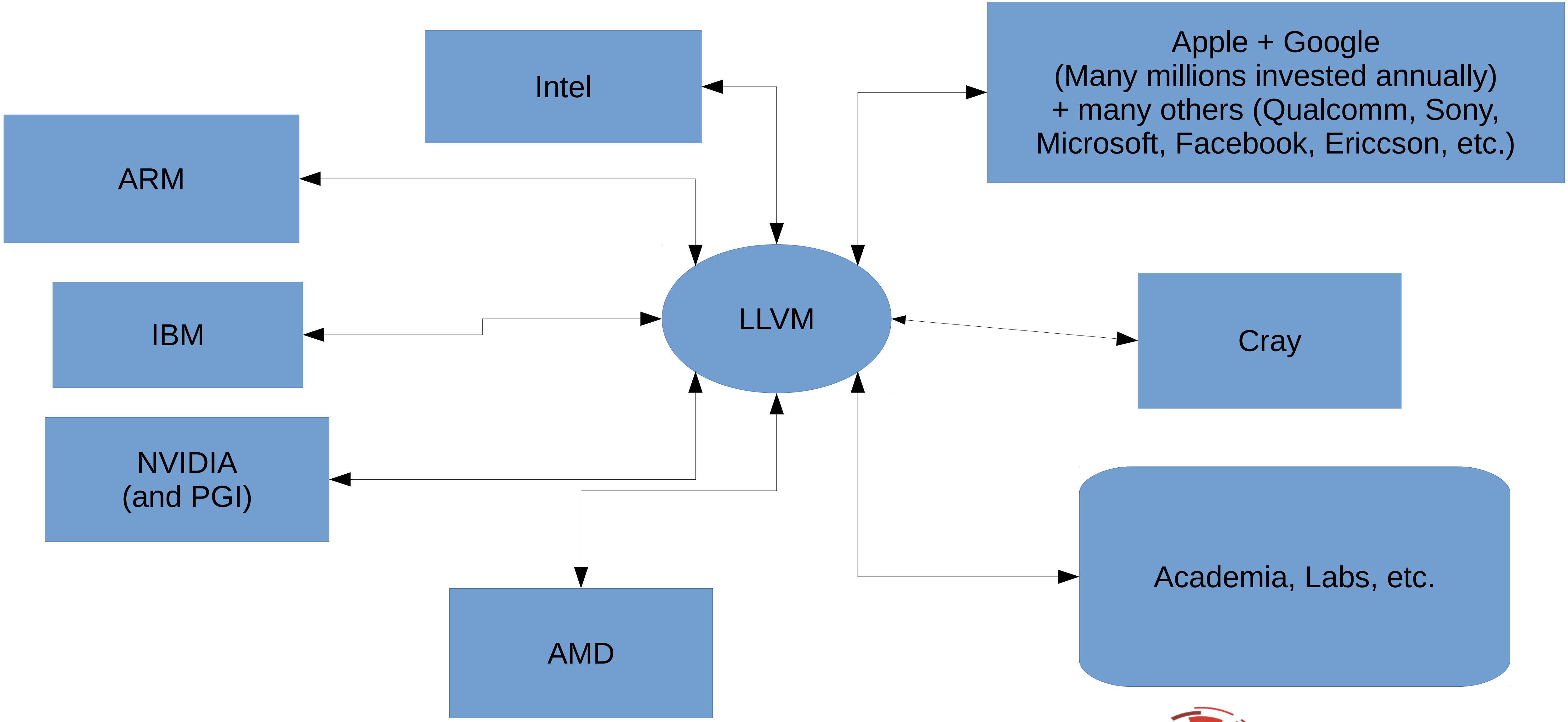
http://llvm.org/devmtg/2016-11/Slides/Amini-Johnson-ThinLTO.pdf

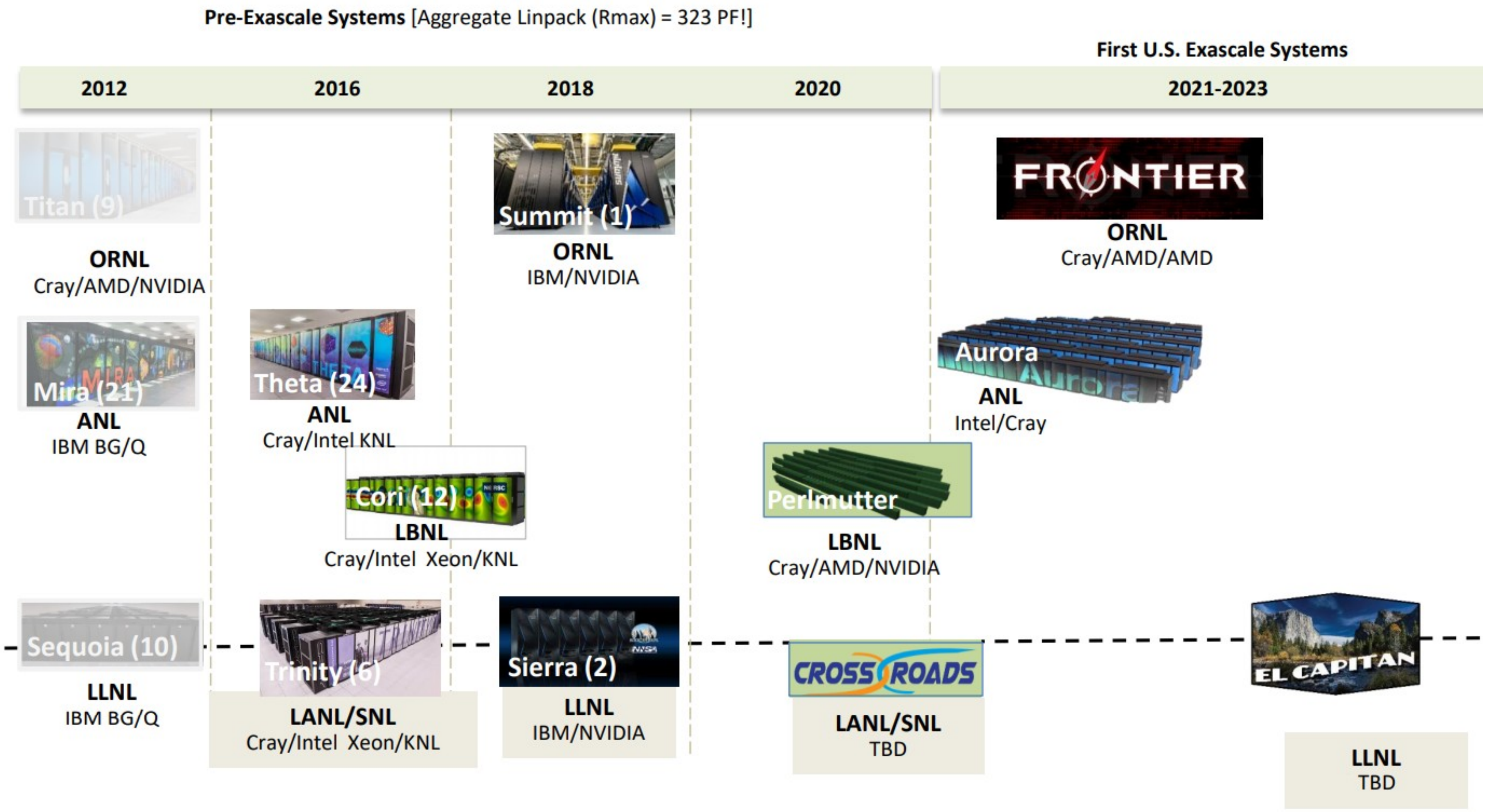
LTO



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

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





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

EXASCALE COMPUTING PROJECT

ECP ST Projects Developing LLVM-Based Technology

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.
- ByFl: 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);
}
```


EXASCALE COMPUTING PROJECT

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)

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];
}
```


EXASCALE COMPUTING PROJECT

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)

Matrix-Matrix Multiplication

After Transformation

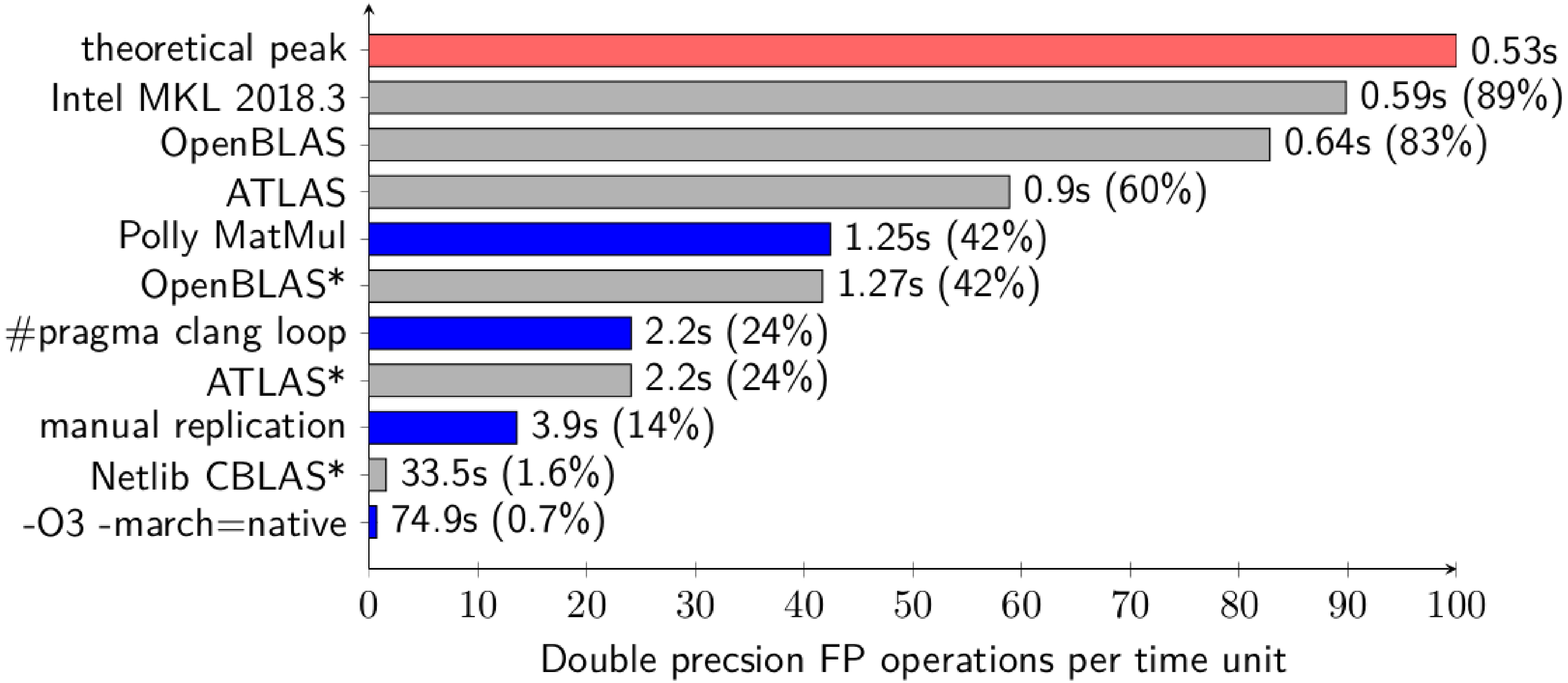
```
double Packed_B[256][2048];
double Packed_A[96][256];
if (runtime check) {
  if (M >= 1)
    for (int c0 = 0; c0 <= floord(N - 1, 2048); c0 += 1)     // Loop j1
      for (int c1 = 0; c1 <= floord(K - 1, 256); c1 += 1) { // Loop k1

        // 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 */
}
```


EXASCALE COMPUTING PROJECT

Loop-Optimization Pragmas and Infrastructure (POC: Michael Kruse, ANL)

Matrix-Matrix Multiplication

Execution Speed



* Pre-compiled from Ubuntu repository

EXASCALE COMPUTING PROJECT

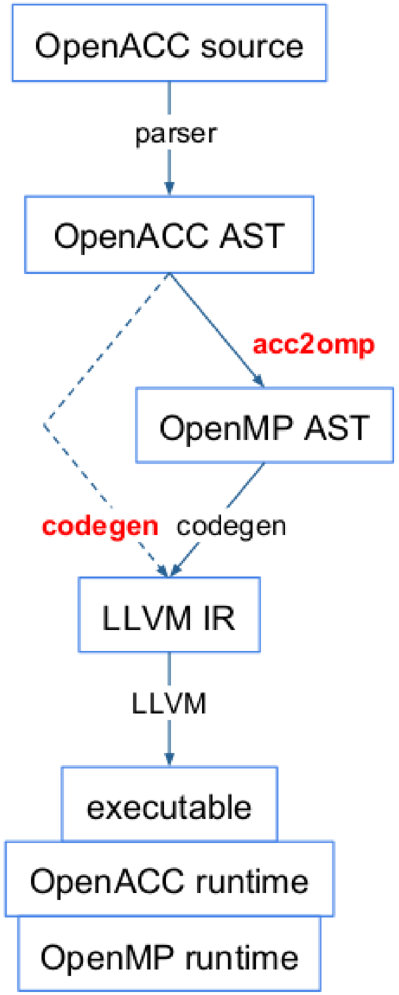
What To Do With OpenACC Code?

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)

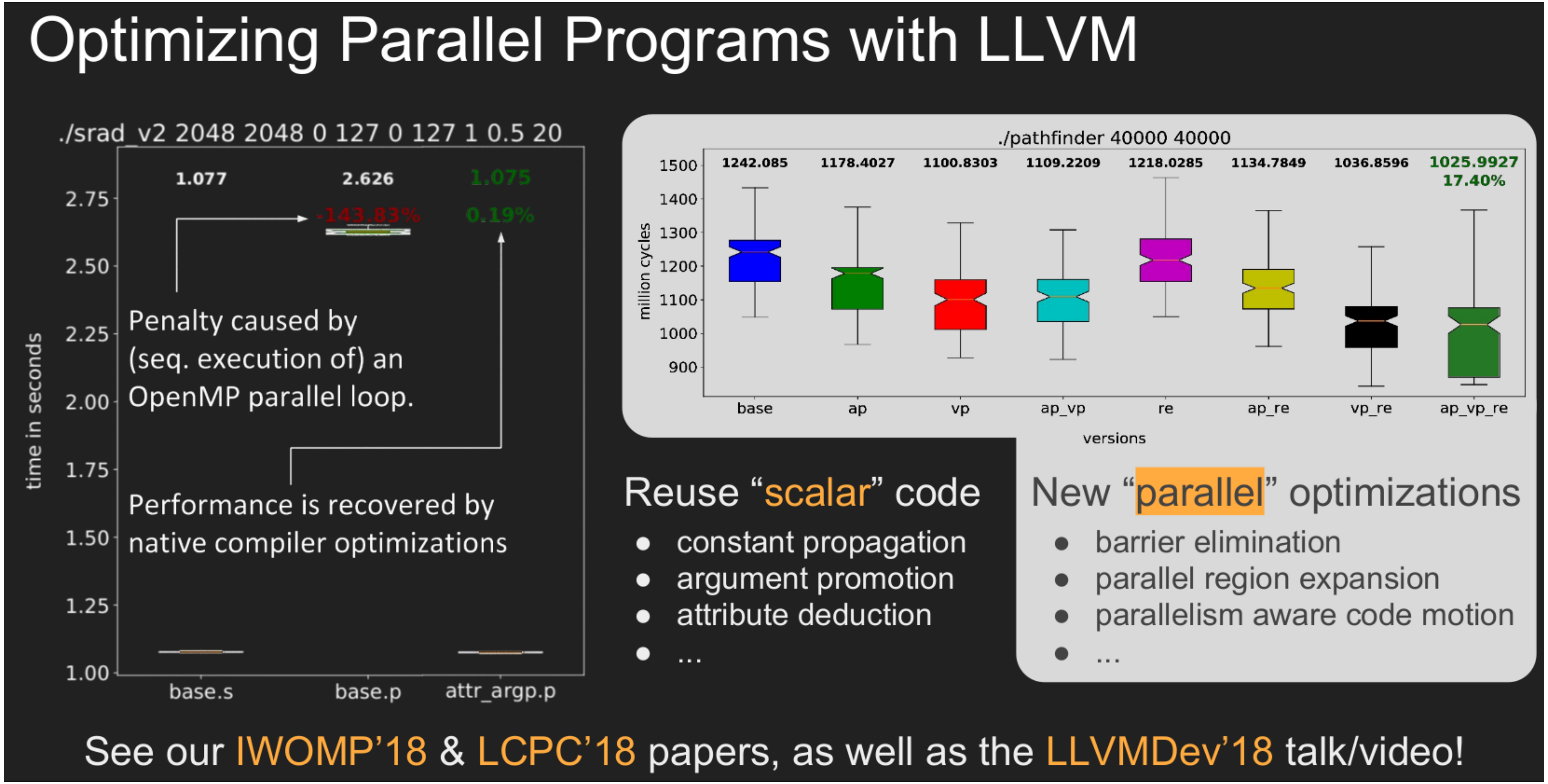
What
- Develop production-quality, standard-conforming traditional OpenACC compiler and runtime support by extending Clang and LLVM
- Enable research and development of source-level OpenACC tools
  - Design compiler to leverage Clang/LLVM ecosystem extensibility
  - E.g., Pretty printers, analyzers, 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



4

EXASCALE COMPUTING PROJECT

Optimization of Parallel Programs (OpenMP and Similar) (POC: Johannes Doerfert, ANL)



EXASCALE COMPUTING PROJECT

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
  - Scalabale 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.