LLVM and Clang:
Advancing Compiler Technology

FOSDEM'11 - Feb 5, 2011

What is the LLVM Umbrella Project?

Language independent optimizer and code generator

- Many optimizations, many targets, generates great code

Clang C/C++/Objective-C front-end

- Designed for speed, reusability, compatibility with GCC quirks

Debuggers, “binutils”, standard libraries

- Providing pieces of a low-level toolchain, with many advantages

Applications of LLVM

- OpenGL, OpenCL, Python, Ruby, etc, even RealBasic and Cray Fortran

LLVM/Clang are Open Source with a BSD-like License!

http://llvm.org/

Why new compilers?

Existing open source C compilers have stagnated!

- Based on decades old code generation technology
- Aging code bases: difficult to learn, hard to change substantially
- Not modular, can't be reused in many other applications
- Keep getting slower with every release

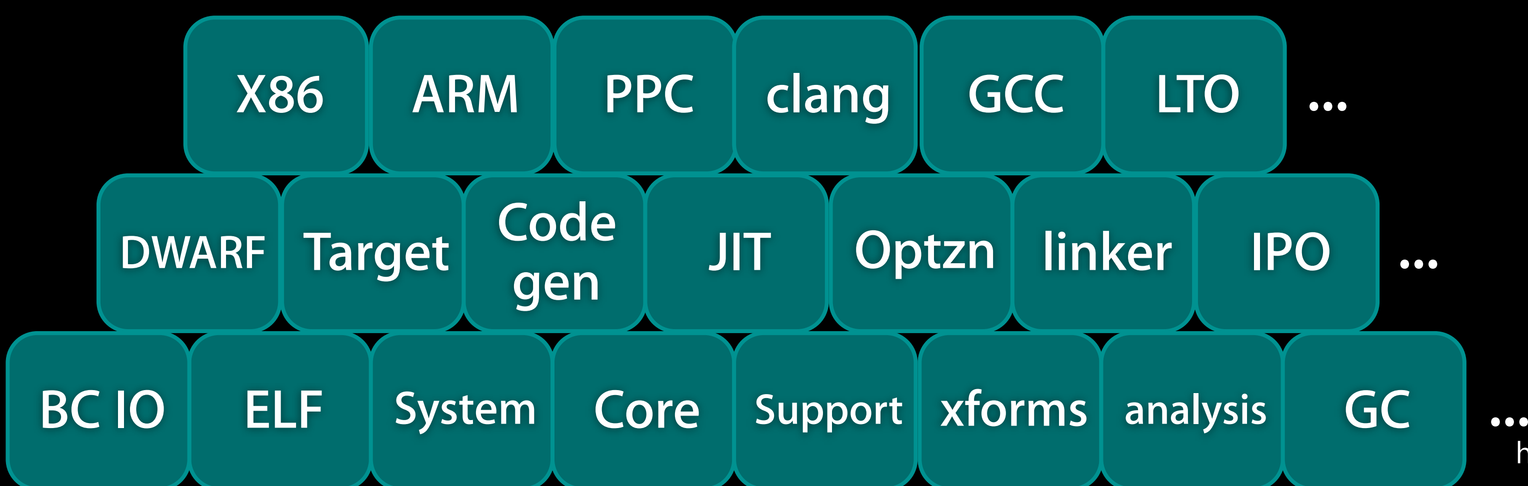
- What I want:
  - A set of production-grade reusable libraries
  - ... which implement the best known techniques
  - ... which focus on compile time
  - ... and performance of the generated code

- Ideally support many different languages and applications!

LLVM Vision and Approach

- Primary mission: **build a set of modular compiler components**:
  - **Reduces the time & cost** to construct a particular compiler
    - A new compiler = glue code plus any components not yet available
  - Components are **shared across different compilers**
    - Improvements made for one compiler benefits the others
  - Allows choice of the **right component for the job**
    - Don't force "one true register allocator", scheduler, or optimization order

- **Secondary mission: Build compilers that use these components**
  - ... for example, an amazing C compiler



http://llvm.org/

LLVM Code Generator Highlights

Approachable C++ code base, modern design, easy to learn

- Strong and friendly community, good documentation

Language and target independent code representation

- Very easy to generate from existing language front-ends
- Text form allows you to write your front-end in perl if you desire

Modern code generator:

- Supports both JIT and static code generation
- Much easier to retarget to new chips than GCC
- Many popular targets supported:
  - X86, ARM, PowerPC, SPARC, Alpha, MIPS, Blackfin, CellSPU, MBlaze, MSP430, XCore, etc.

http://llvm.org/docs/

Example Application: LLVM + OpenGL

Colorspace Conversion

- Code to convert from one color format to another:
  - e.g. BGRA 444R to RGBA 8888
  - Hundreds of combinations, importance depends on input

```
for each pixel {
  switch (infmt) {
  case RGBA5551:
    R = (*in >> 11) & C
    G = (*in >> 6) & C
    B = (*in >> 1) & C
  ... }
  switch (outfmt) {
  case RGB888:
    *outptr = R << 16 |
              G << 8 ...
  }
}
```

Run-time specialize

```
for each pixel {
  R = (*in >> 11) & C;
  G = (*in >> 6) & C;
  B = (*in >> 1) & C;
  *outptr = R << 16 |
            G << 8 ...
}
```

Compiler optimizes shifts and masking

- Speedup depends on src/dest format:
  - 5.4x speedup on average, 19.3x max speedup: (13.3MB/s to 257.7MB/s)

OpenGL Pixel/Vertex Shaders

- Small program run on each vertex/pixel, provided at run-time:
  - Written in one of a few high-level graphics languages (e.g. GLSL)
  - Executed millions of times, extremely performance sensitive
- Ideally, these are executed on the graphics card:
  - What if hardware doesn't support some feature? (e.g. laptop gfx)
    - **Interpret or JIT on main CPU**

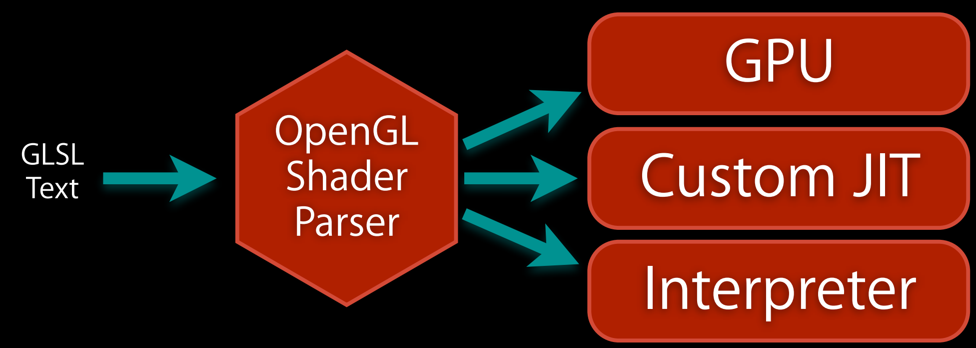
```
void main() {
 vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
 vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
 vec3 lightVec   = normalize(LightPosition - ecPosition);
 vec3 reflectVec = reflect(-lightVec, tnorm);
 vec3 viewVec    = normalize(-ecPosition);
 float diffuse   = max(dot(lightVec, tnorm), 0.0);
 float spec      = 0.0;
 if (diffuse > 0.0) {
     spec = max(dot(reflectVec, viewVec), 0.0);
     spec = pow(spec, 16.0);
 }
 LightIntensity = DiffuseContribution * diffuse +
                  SpecularContribution * spec;
 MCposition     = gl_Vertex.xy;
 gl_Position    = ftransform();
}
```

OpenGL Pixel/Vertex Shader (GLSL)

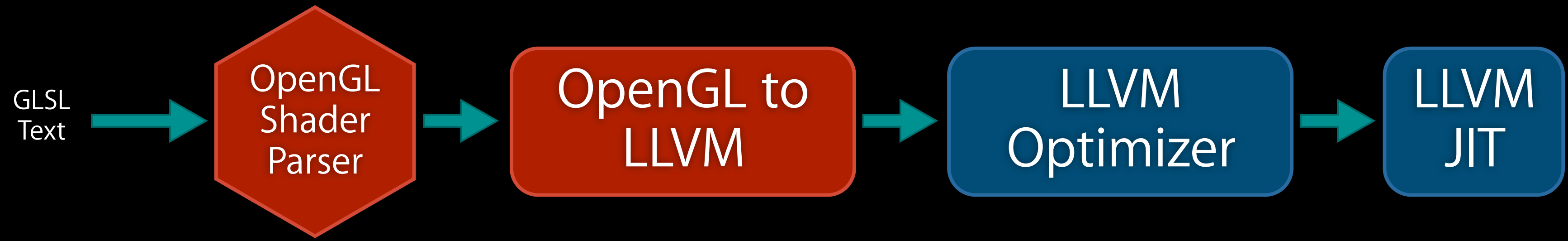
OpenGL Implementation Before LLVM

- Custom JIT for X86-32 and PPC-32:
  - Very simple codegen: pasted chunks of AltiVec or SSE code
  - Little optimization across operations (e.g. scheduling)
  - Very fragile, hard to understand and change (hex opcodes)

- Interpreter:
  - JIT didn't support all OpenGL features: fallback to interpreter
  - Interpreter was very slow, 100x or worse than JIT



OpenGL JIT Built with LLVM Components



- At runtime, build LLVM IR for program, optimize, JIT:
  - Result supports any target LLVM supports
  - Generated code is as good as an optimizing static compiler
- OpenGL benefits from LLVM optimizer/codegen improvements

How does the "OpenGL to LLVM" stage work?

Detour: Structure of an Interpreter

- Simple opcode-based dispatch loop:

```
while (...) {
  ...
  switch (cur_opcode) {
  case dotproduct:     result = opengl_dot(lhs, rhs); break;
  case texturelookup: result = opengl_texlookup(lhs, rhs); break;
  case ...
```

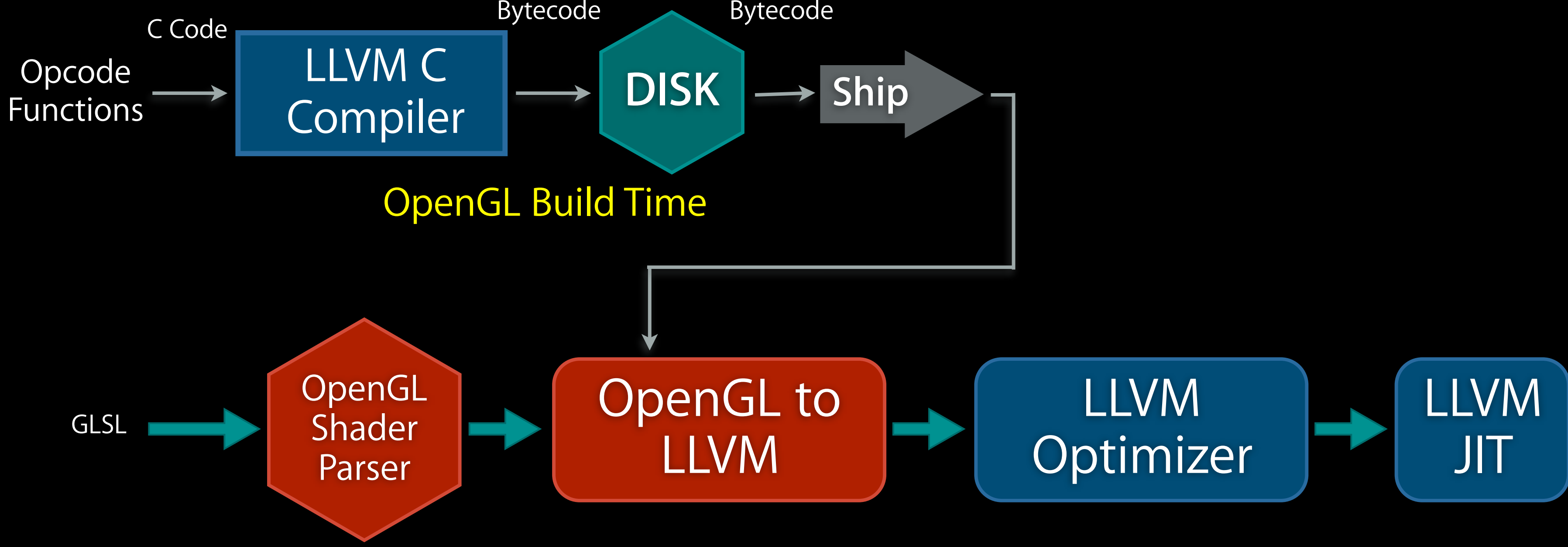
- One function per operation, written in C:

```
double opengl_dot(vec3 LHS, vec3 RHS) {
 #ifdef ALTIVEC
   ... altivec intrinsics ...
 #elif SSE
   ... sse intrinsics ...
 #else
   ... generic c code ...
  #endif
}
```

Key Advantage of an Interpreter:
Easy to understand and debug, easy to write each operation (each operation is just C code)

- In a high-level language like GLSL, ops can be hundreds of LOC

OpenGL to LLVM Implementation

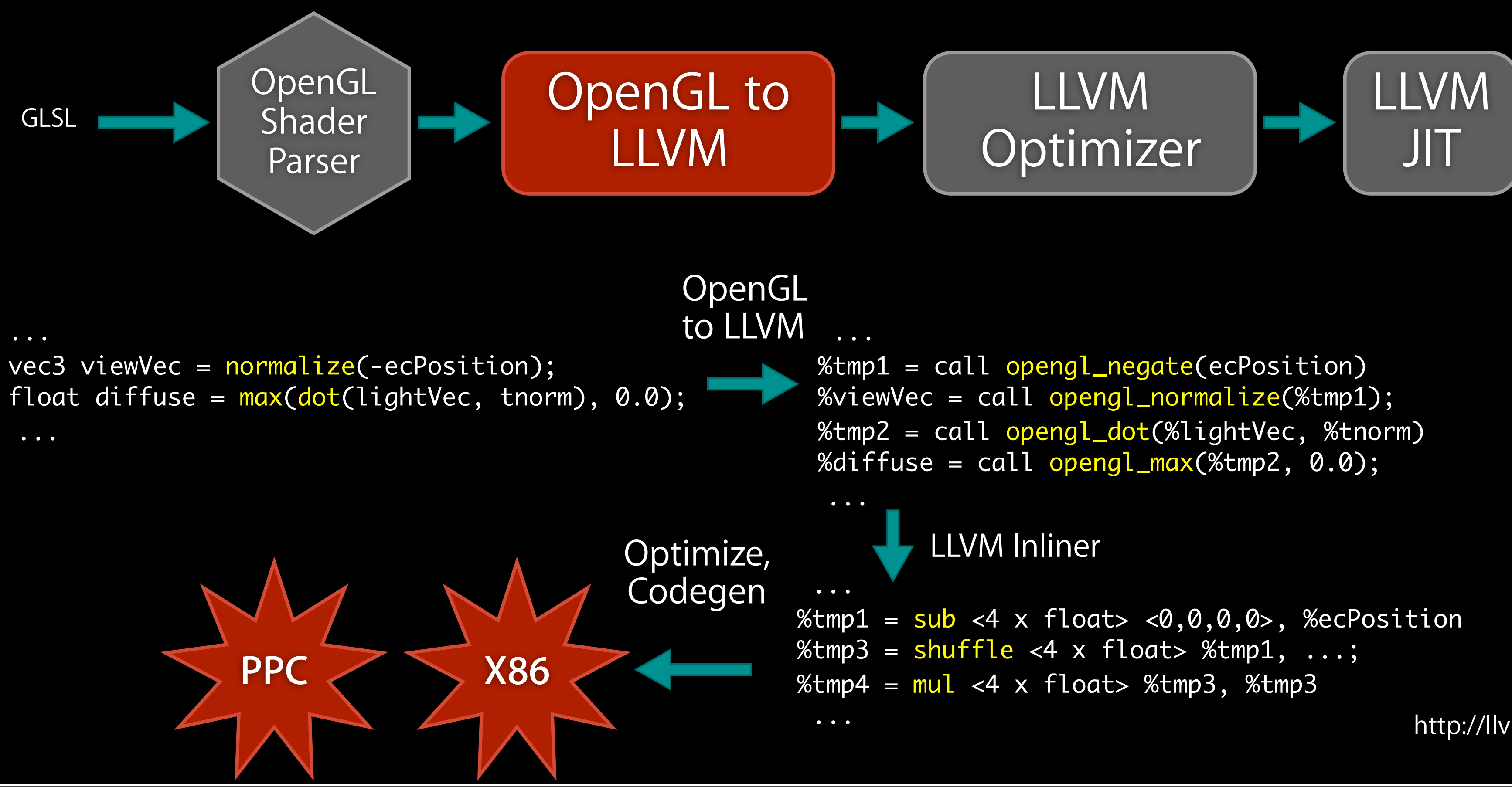


- At OpenGL build time, compile each opcode to LLVM bytecode:
  - Same code used by the interpreter: easy to understand/change/optimize

http://llvm.org/

OpenGL to LLVM: At runtime

1. Translate OpenGL AST into LLVM call instructions: one per operation
2. Use the LLVM inliner to inline opcodes from precompiled bytecode
3. Optimize/codegen as before



http://llvm.org/

Benefits of this Approach

- Each opcode is written/debugged for a simple interpreter
  - as standard C code
- Retains all advantages of an interpreter:
  - debug-ability, understandability, etc
- Easy to make algorithmic changes to opcodes
- Great performance!

Lots of Other Applications

- OpenCL: a GPGPU language, with most vendors using LLVM
- Dynamic Languages: Unladen Swallow, Rubinious, MacRuby
- llvm-gcc 4.2 & DragonEgg
- Cray Cascade Fortran Compiler
- vmkit: Java and .NET VMs
- Haskell, Mono, LDC, Pure, Roadsend PHP, RealBasic
- IOQuake3 for real-time raytracing of Quake!

http://llvm.org/Users.html

Clang Compiler

Clang Goals

- Unified parser for C-based languages
  - Language conformance (C, Objective-C, C++)
  - Useful error and warning messages

- Library based architecture with finely crafted API's
  - Useable and extensible by mere mortals
  - Reentrant, composable, replaceable

- Multi-purpose
  - Indexing, static analysis, code generation
  - Source to source tools, refactoring

http://clang.llvm.org/

Clang Goals #2

- High performance!
  - Low memory footprint, fast compiles
  - Support lazy evaluation, caching, multithreading
  - get the compiler out of the way during development

- Highly Compatible with GCC
  - Supports almost all the arcane, but useful, GCC extensions
  - GCC Inline ASM and CPU built-ins / intrinsics supported
  - Aim for drop-in replacement where reasonable

Clang Compiler Status

- C, Objective-C, and C++ support are production quality
  - Clang has successfully compiled millions of lines of C/C++/Objective-C code
  - Can bootstrap itself, build Boost, Mozilla, and many other "compiler busters"
  - Builds a working FreeBSD base system
  - Interesting tools starting to be built on it

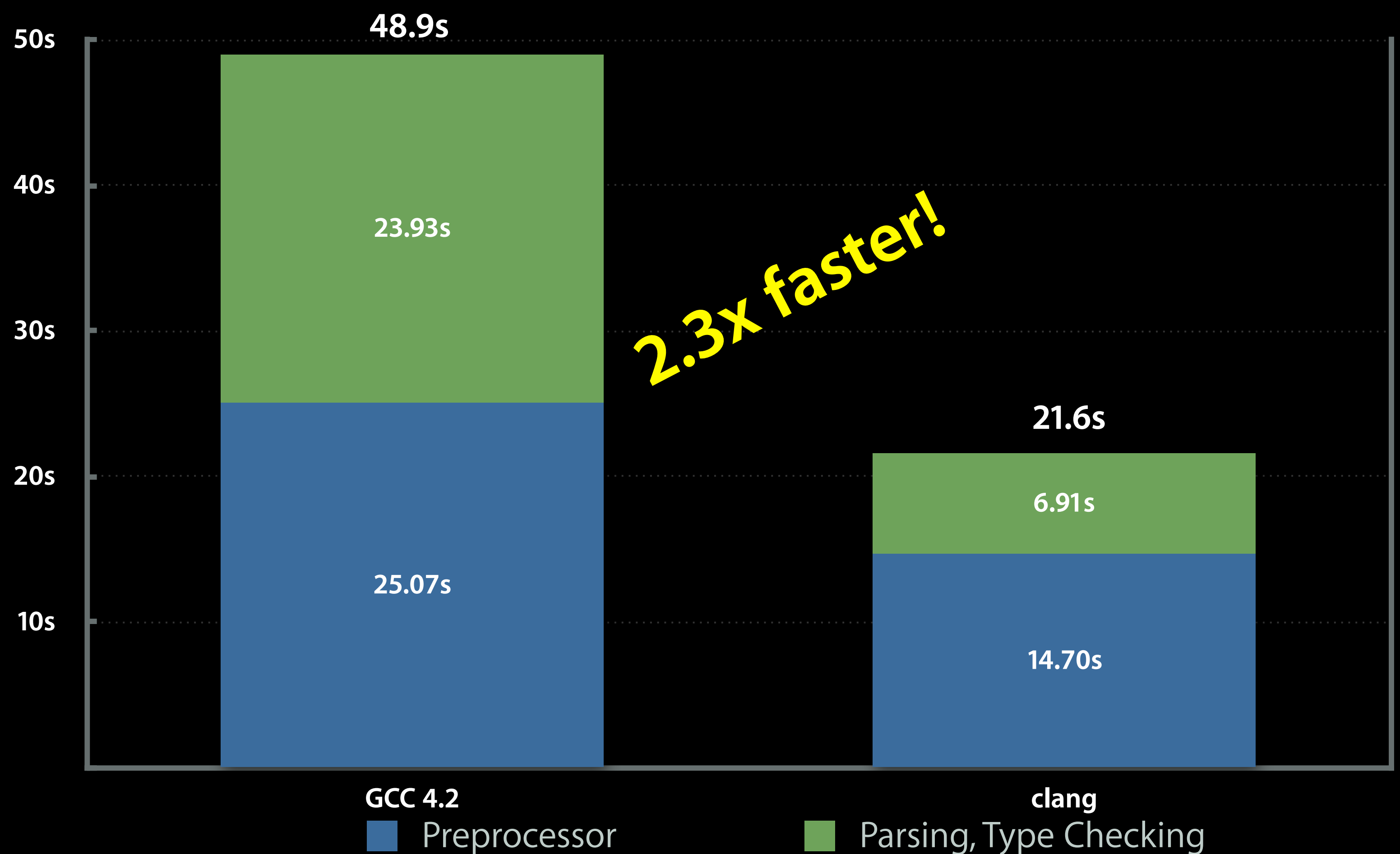
- Common stumbling blocks migrating from GCC to Clang:
  - C89 vs C99 inlining differences
  - Bugs in G++'s template implementation
  - http://clang.llvm.org/compatibility.html

- Work is progressing on MSVC compatibility and C++'0x support

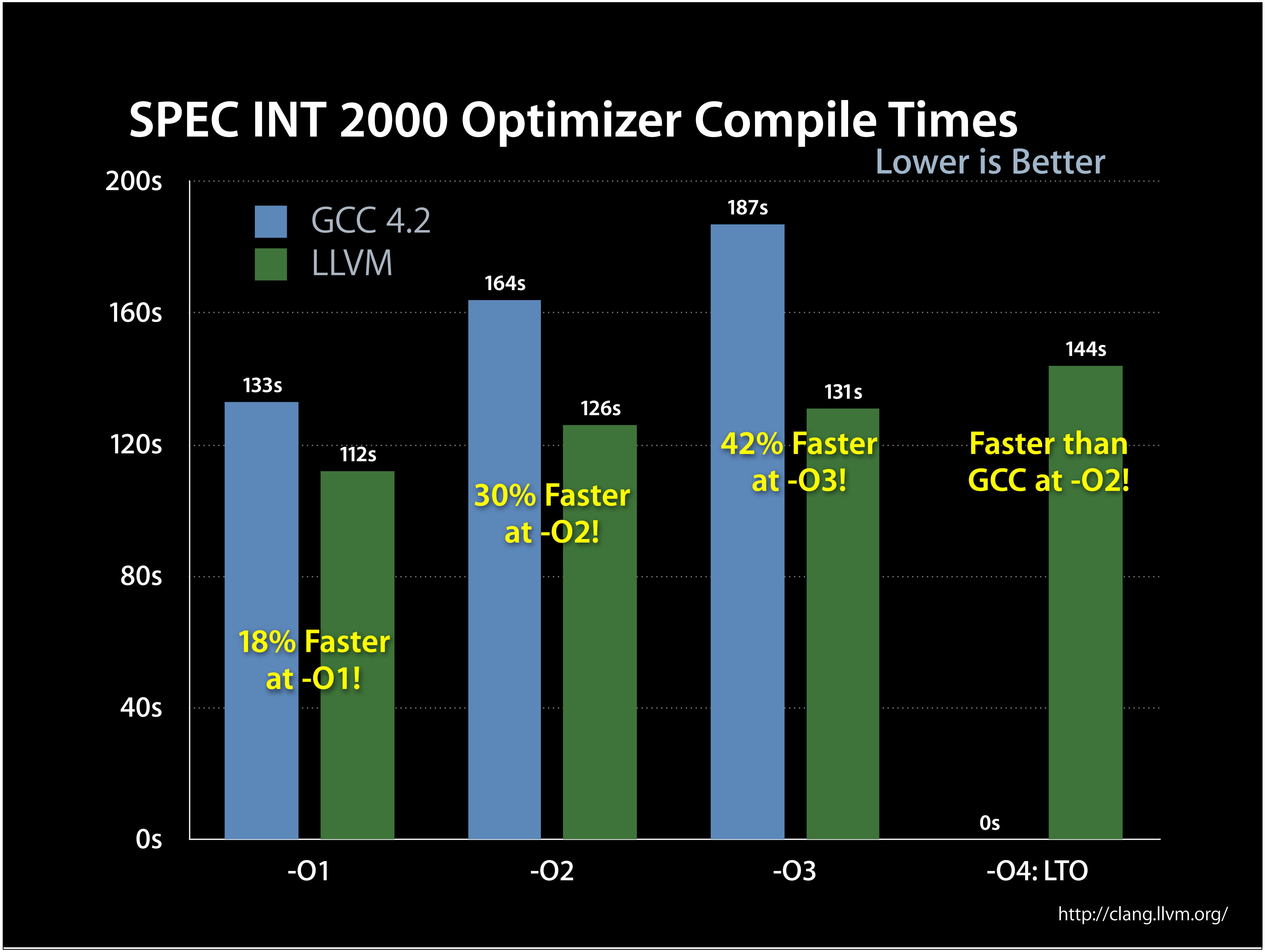
Shockingly fast and memory efficient, much better user experience!

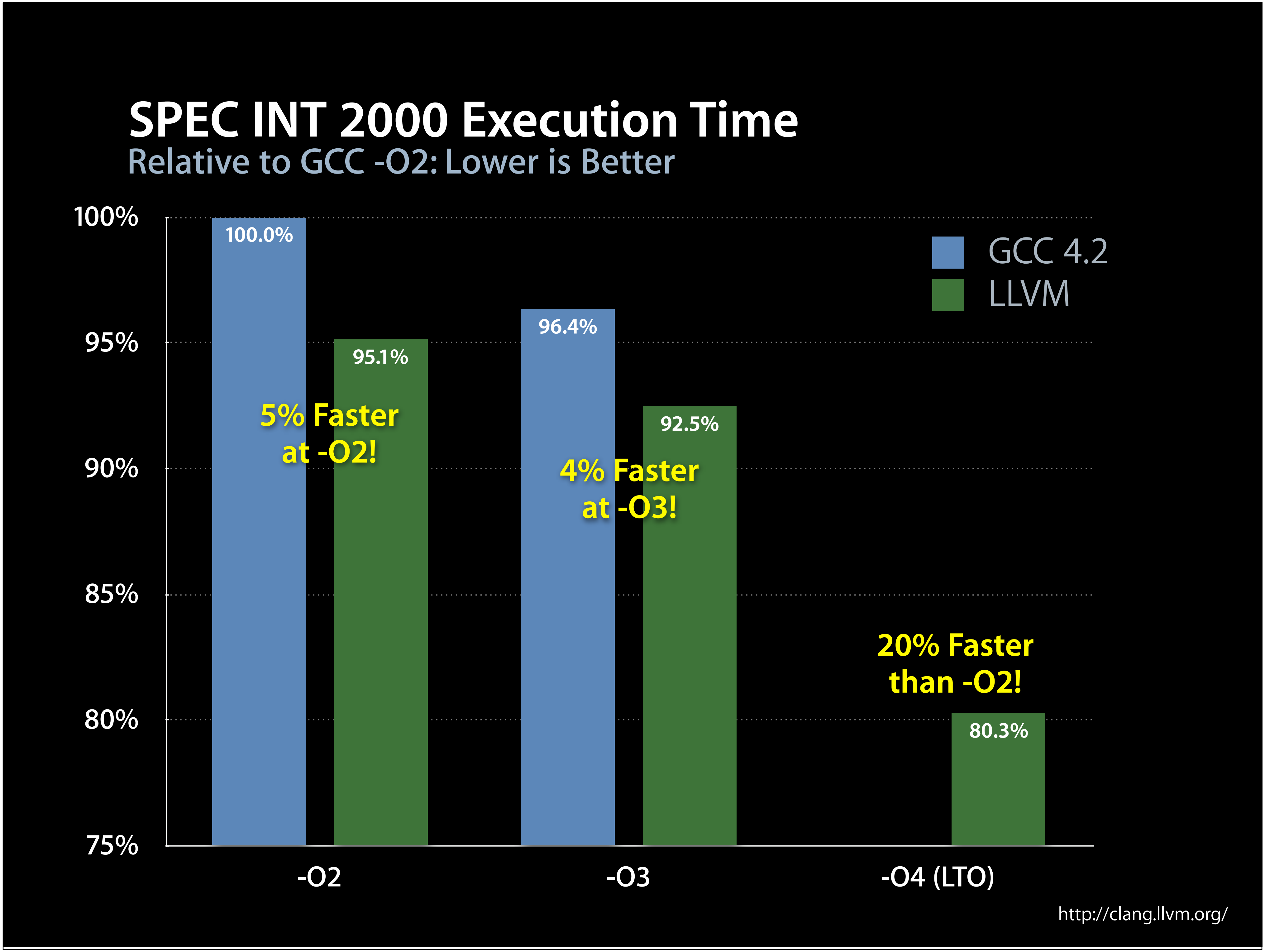
Compile Time Comparison: Front-end

PostgreSQL: a medium sized C project: 619 C Files in 665K LOC, excluding headers



http://clang.llvm.org/performance.html


SPEC INT 2000 Optimizer Compile Times
Lower is Better
200s
160s
120s
80s
40s
0s
GCC 4.2
LLVM
133s
112s
164s
126s
187s
131s
0s
144s
18% Faster at -O1!
30% Faster at -O2!
42% Faster at -O3!
Faster than GCC at -O2!
-O1
-O2
-O3
-O4: LTO
http://clang.llvm.org/


SPEC INT 2000 Execution Time
Relative to GCC -O2: Lower is Better
100%
95%
90%
85%
80%
75%
GCC 4.2
LLVM
100.0%
95.1%
96.4%
92.5%
80.3%
5% Faster at -O2!
4% Faster at -O3!
20% Faster than -O2!
-O2
-O3
-O4 (LTO)
http://clang.llvm.org/

User Experience: Diagnostics

```
$ clang implicit-def.c -std=c89
implicit-def.c:6:10: warning: implicit declaration of function 'X'
  return X();
         ^
```

```
   struct A { int X; } someA;
   int func(int);

   int test1(int intArg) {
5:   intArg += *(someA.X);
6:   return intArg + func(intArg ? ((someA.X + 40) + someA) / 42 + someA.X : someA.X));
   }
```

```
% gcc-4.2 t.c
t.c: In function 'test1':
t.c:5: error: invalid type argument of 'unary *'
t.c:6: error: invalid operands to binary +
```

http://clang.llvm.org/

User Experience: “Expressive” Diagnostics

- Other Features:
  - std::string instead of std::basic_string<char, std::char_traits<char>, std::allocator<char> >
  - #pragma control over diagnostics
  - Doesn’t “pretty print” expressions back out at you

```
% clang test.c
t.c:5:13: error: indirection requires pointer operand ('int' invalid)
  intArg += *(someA.X);
            ^~~~~~~~~~
t.c:6:49: error: invalid operands to binary expression ('int' and 'struct A')
  return intArg + func(intArg ? ((someA.X+40) + someA) / 42 + someA.X : someA.X));
                                 ~~~~~~~~~~~~ ^ ~~~~~
```

```
% gcc-4.2 t.c
t.c: In function 'test1':
t.c:5: error: invalid type argument of 'unary *'
t.c:6: error: invalid operands to binary +
```

http://clang.llvm.org/

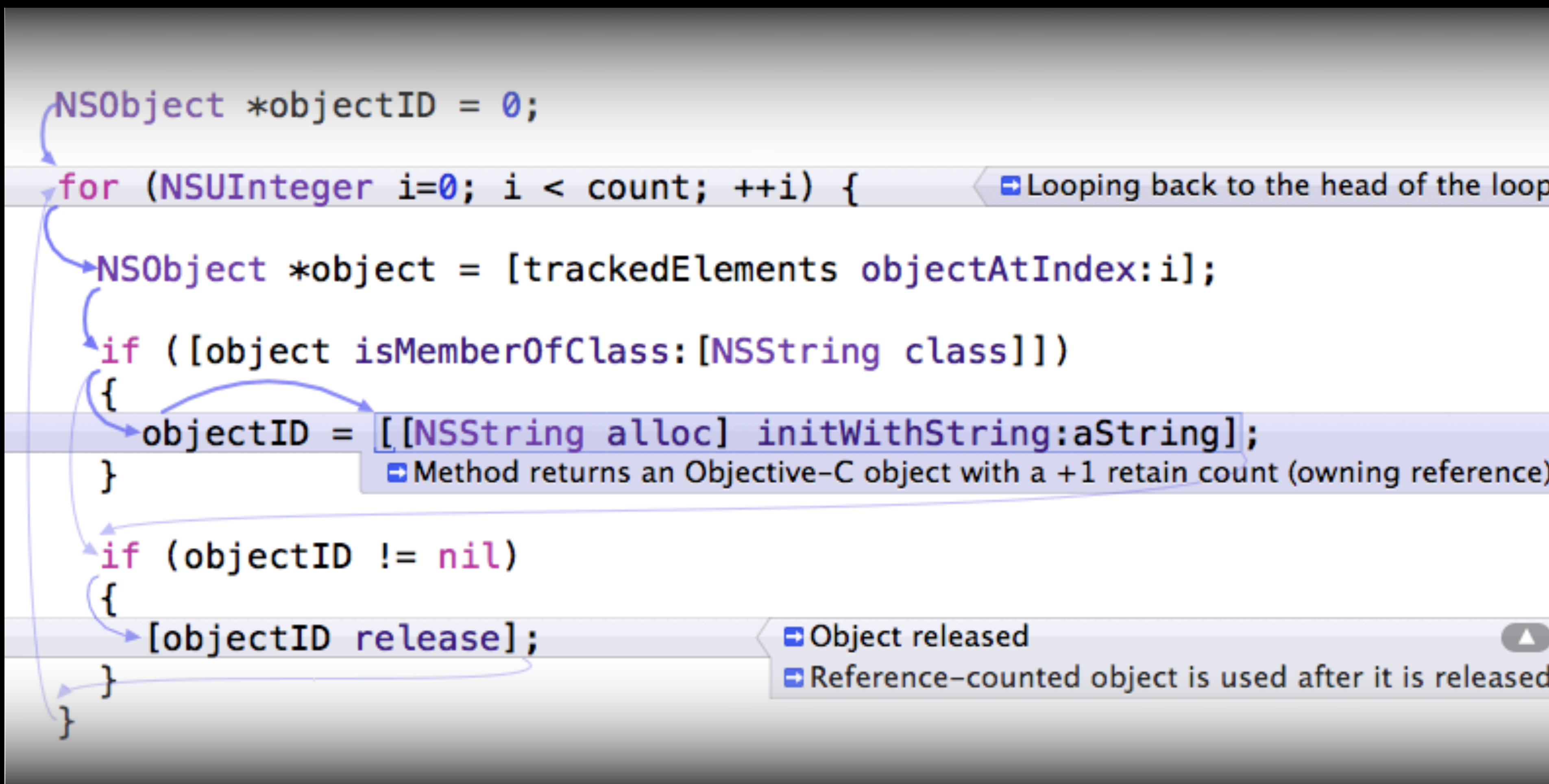
Other Improvements

```
$ g++-4.2 t.cpp
t.cpp:12: error: no match for 'operator=' in 'str = vec'

$ clang t.cpp
t.cpp:12:7: error: incompatible type assigning 'vector<Real>', expected
'std::string' (aka 'class std::basic_string<char>')
  str = vec;
      ^ ~~~


t.c:48:7: error: invalid operands to binary expression ('int' and 'struct A')
   X = MAX(X, *Ptr);
       ^~~~~~~~~~~~


t.c:43:24: note: instantiated from:
 #define MAX(A, B) ((A) > (B) ? (A) : (B))
                    ~~~ ^ ~~~
```

http://clang.llvm.org/diagnostics.html

Clang Static Analyzer

- Automatically finds and reports bugs in your code
- Uses deep analysis techniques to explore things that testing misses



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

Other Notable LLVM Projects

- MC: Machine Code slicing and dicing
  - Assemblers, disassemblers, object file processing
- LLDB: Low Level Debugger — http://lldb.llvm.org/
  - Command-line debugger
  - Reuses Clang parser, LLVM JIT, MC disassemblers
  - Great support for C++, and multithreaded apps
- libc++: C++ standard runtime library — http://libcxx.llvm.org/
  - Full support for C++'0x
  - "No compromises" performance

http://llvm.org/devmtg/2010-11/

LLVM and Clang

- Compiler infrastructure built with reusable components
  - Bringing compiler techniques to new interesting problems
- LLVM: flexible optimizer and code generator
  - Fast compiles, great generated code
  - Supports many targets
  - Reusable in nontraditional contexts
- Clang: C/ObjC/C++ front-end
  - Multiple times faster than other compilers
  - Great end-user features (e.g. warnings/errors)
  - Platform for new source level tools

Come join us at:

http://llvm.org

http://clang.llvm.org

http://llvm.org/