LLVM - 2.0 and beyond!

http://llvm.org/

Chris Lattner
clattner@apple.com

Google Tech Talk
July 27, 2007

What is the LLVM Project?

Two primary components:

- Mid-Level Optimizer and Code Generator
  - Standard Suite of SSA-based optimizations
    - Scalar optimizations, loop optimizations, load/store xforms, etc.
  - Codegen support for many targets:
    - X86, X86-64, PPC, PPC64, ARM, Thumb, SPARC, Alpha, IA64, MIPS
- llvm-gcc front-end for C/C++/ObjC/Ada/FORTRAN/...
  - Based on GCC 4.0, translates from GIMPLE to LLVM IR.
  - Uses GCC frontend with LLVM Optimizers and Codegen
  - Drop in compatibility with GCC: make CC=llvm-gcc

Novel Capabilities of LLVM

Relative to GCC

- Link Time Optimization (cross-file optimization)
  - Includes a full suite of IPO transformations, since LLVM 1.0
  - Integrates with native system linker
- Just-In-Time (JIT) code generation
  - Can optionally optimize and generate code on the fly
  - Used by Apple (OpenGL stack) and Adobe (After Effects CS3)
    - ... among others, see http://llvm.org/Users.html
- Easy to learn, work with, and contribute to:
  - Clean and modular design, good support tools (bugpoint)
  - Strong and friendly community, good documentation

Talk Outline

LLVM - 2.0 and beyond!

- What's new in LLVM 2.0 (May 23, 2007)
  - New features, optimizations, major changes
- What is coming next?
  - LLVM 2.1 (~Sep 2007)
  - llvm-gcc 4.2
  - "clang" - New C front-end

What’s new in LLVM 2.0

New features, optimizations, major changes

New Features in LLVM 2.0 (vs 1.9)

- **Missing ELF features added:**
  - visibility control
  - Extern weak linkage
  - Thread Local Storage: __thread
  - Symbol Aliases
- **Other missing features added:**
  - #pragma pack
  - alignment info for loads/stores
  - soft float support
  - Precompiled Headers (PCH)
- **Missing X86 features added:**
  - MMX support
  - linux/i386 PIC support
  - regparm parameters
- **Still missing in LLVM 2.0:**
  - C++ Zero Cost EH (LLVM 2.1)
  - long double (LLVM 2.1 or 2.2)
  - Nested function trampolines (?)
  - __builtin_apply (?)

Can now build many large software packages out of the box: mozilla, Qt, etc...

Other LLVM 2.0 user-visible improvements

- Optimizer / Codegen Improvements:
  - Can now promote simple unions to registers (important for vectors)
  - Better register coallescing: avoid coallescing that leads to spilling
  - Simple rematerialization support in the register allocator
  - Register pressure optimizations in the scheduler
  - Improved switch statement lowering
- Other new stuff:
  - Compile-time speedup in several places
  - Better llvm-gcc support for Ada (which generates "aggressive" GIMPLE)
  - MSIL backend (transforms LLVM IR into unsafe MSIL)

LLVM 2.0 Internal IR Changes

- LLVM Intermediate Representation is stored in two kinds of files:
  - .ll file - Human readable/writable file, looks like assembly language
  - .bc file - Binary file, very compact and fast to read/write

- "2.0" allowed us to break backwards compatibility with 1.x IR files:
  - LLVM 1.9 transparently upgrades .ll files and .bc files from LLVM 1.3 - 1.9
  - LLVM 2.0 requires use of llvm-upgrade tool to upgrade llvm 1.9 .ll file
  - '2.0' allows major IR changes, that would be difficult to 'auto-upgrade'

- Two major IR changes:
  - Signless types
  - Arbitrary-precision integers

LLVM 2.0 IR: Signless Types

- The LLVM 1.9 IR uses integer type system very similar to C:
  - signed vs unsigned 8/16/32/64 bit integers
- The LLVM 2.0 IR uses integer types similar to CPUs:
  - 8/16/32/64 bits, with sign stored in operators (e.g. sdiv vs udiv)

- Advantages:
  - Smaller IR: don't need "noop" conversions (e.g. uint <--> sint)
  - More explicit operations (e.g. 'sign extend' instead of 'cast')
  - Redundancy elimination: (actually happens during strength reduction)

```
int foo(unsigned A) {
  int B = (int)A;
  unsigned C = A + 4;
  int D = B + 4;
  int E = (int)C;
  return D-E;
}
```

C Code

```
int %foo(uint %A) {
  %B = cast uint %A to int
  %C = add uint %A, 4
  %D = add int %B, 4
  %E = cast uint %C to int
  %F = sub int %D, %E
  ret int %F
}
```

LLVM 1.9

```
i32 @foo(i32 %A) {

  %C = add i32 %A, 4
  %D = add i32 %A, 4

  %F = sub i32 %D, %C
  ret i32 %F
}
```

LLVM 2.0

LLVM 2.0 IR: Arbitrary precision integers

- LLVM 2.0 allows arbitrary fixed width integers:
  - i2, i13, i128, i1047, i12345, etc
- Primarily useful to EDA / hardware synthesis business:
  - An 11-bit multiplier is significantly cheaper/smaller than a 16-bit one
  - Can use LLVM analysis/optimization framework to shrink variable widths
  - Patch available that adds an attribute in llvm-gcc to get this
- Implementation impact of arbitrary width integers:
  - Immediates, constant folding, intermediate arithmetic simplifications
  - New "APInt" class used internally to represent/manipulate these
  - Makes LLVM more portable, not using uint64_t everywhere for arithmetic

What’s next?

LLVM 2.1, llvm-gcc 4.2, clang CFE

Coming Soon - LLVM 2.1: September '07

- LLVM release cycle varies from 3-6 months
  - Emphasize incremental development, discourage development branches

- Major new LLVM 2.1 features will likely include:
  - C++ Zero-Cost DWARF Exception handling support for linux/x86
  - Several improvements in the optimizer
  - Compile time improvements (20%+ faster optimizer at -O3 in some cases)

- LLVM 2.1 code freeze is on Sep 12 '07 - get your feature in soon! :)

Coming Soon - llvm-gcc 4.2

Update llvm-gcc from GCC 4.0 to GCC 4.2

- Bring new GCC 4.2 front-end features to LLVM:
  - OpenMP, better warning control, visibility support
  - Many FORTRAN and Ada improvements
  - New recursive descent C/ObjC parser

- Current Status:
  - Work just started in mid-July, already making fast progress
  - Can build front-end, libgcc, etc, but cannot bootstrap yet
  - Current plan is for this to be ready for LLVM 2.2.

LLVM C Family Frontend “clang”

http://clang.llvm.org/

Motivation: Why a new front-end?

- GCC doesn't service the diverse need of an IDE
  - Indexing - scoped variable uses and defs: 'jump to definition' 'doxygen'
  - Source analysis - 'automatic bug finding'
  - Refactoring - 'Rename variable' 'pull code into a new function'
  - Other source-to-source transformation tools, like 'smart editing'
  - Yes, it does support compiling :-)

- GCC does not preserve enough source-level information
  - Full column numbers, it implicitly folds/simplifies trees as it parses, etc

- GCC's front-end is difficult to work with
  - Learning curve too steep for many developers
  - Implementation and politics limit innovation

- GCC's front-end is very slow and memory hungry

Goals

- Unified parser for C-based languages
  - Language conformance (C, Objective C, C++) & GCC compatibility
  - Good diagnostics
- Library based architecture with finely crafted C++ API's
  - Useable and extensible by mere mortals
  - Reentrant, composable, replaceable
- Multi-purpose
  - Indexing, static analysis, code generation
  - Source to source tools, refactoring
- High performance!
  - Low memory footprint, fast compiles
  - Support lazy evaluation, caching, multithreading

Non Goals

- Support for non-C based languages
  - No plans for Java, Ada, FORTRAN, etc
  - These languages can reuse some code, but the ASTs are C-specific
  - Separate front-end projects could do this if someone steps up to do it

- Obsoleting GCC (or llvm-gcc)
  - not pragmatic, we respect GCC's ubiquity
  - our goals are fundamentally different than GCC
  - our contributors set our priorities, GCC's contributors set theirs
    - GCC/LLVM will always have distinct (but partially overlapping) strengths

Introducing "clang": http://clang.llvm.org/

- A simple driver, with (some) GCC compatible options:

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

- Performance analysis options (-Eonly, -parse-noop, -stats)

```
$ time clang -parse-noop INPUTS/carbon_h.c
real  0m0.204s
user  0m0.138s
sys   0m0.047s
```

- Several useful features built in:

```
$ clang -parse-ast-print madd.c
typedef float V;
V foo(V a, V b) {
   return a + b * a;
}
```

Pretty Printer from ASTs

```
$ cat testcase.c
typedef union <bad> __mbstate_t;  // \
   expected-error: {{not really here}}
$ clang -parse-ast-check testcase.c
Errors expected but not seen:
  Line 1: not really here
Errors seen but not expected:
  Line 1: expected identifier or '{'
```

Diagnostic Checker (for testsuite)

GCC diagnostics

```
// test.c
struct A { int X; } someA;
int func(int);

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

```
% cc -c test.c

test.c: In function ‘test1’:
test.c:7: error: invalid type argument of ‘unary *’
test.c:8: error: invalid operands to binary +
```

clang “expressive” diagnostics

```
% clang test.c

test.c:7:2: error: indirection requires a pointer operand ('int' invalid)
 *(someA.X);
 ^~~~~~~~~~
test.c:8:48: error: invalid operands to binary expression ('int' and 'struct A')
 return intArg + func(intArg ? ((someA.X + 40) + someA) / 42 + someA.X : someA.X);
                                ~~~~~~~~~~~~~~ ^ ~~~~~
```

```
% cc -c test.c

test.c: In function ‘test1’:
test.c:7: error: invalid type argument of ‘unary *’
test.c:8: error: invalid operands to binary +
```

- Other Features:
  - Retains typedef info: std::string instead of std::basic_string<char, std::char_traits<char>, std::allocator<char> >
  - Diagnostics have unique ID’s (good for internationalization, control, IDE)
  - Fine grained location tracking (even through macro instantiations)

clang status: still very early (July 26, '07)

- Major components:
  - Lexer and preprocessor are done and well tested
  - C Parser is 95% complete
  - Required semantic analysis / type-checking for C is 80% done (errors)
  - "QOI" Semantic analysis (warnings) largely missing
  - Code generation through LLVM: 15% done

- Objective C and C++ support are almost completely missing

Work continues, and there are already many interesting things we can do: are we on the right track?

Compile Time Performance

http://clang.llvm.org/

Three primary scenarios:

- Release builds:
  - Highly optimized -O4 builds

  front-end has little effect*: speed up the optimizer/codegen

- Development builds:
  - "-O0 -g" builds: fast compile, debug, edit cycles

  front-end is critical!

- Source-Level Analysis tools:
  - e.g. indexing: need to keep the index up-to-date as user edits code

  front-end is critical!

Question: Where does a front-end spend its time?

* for uniprocessor builds, more later

Anatomy of a typical GUI app on the mac

- Contains a few dozen or few hundred source files (C or Objective C)
  - Each file may be a few thousand lines of code

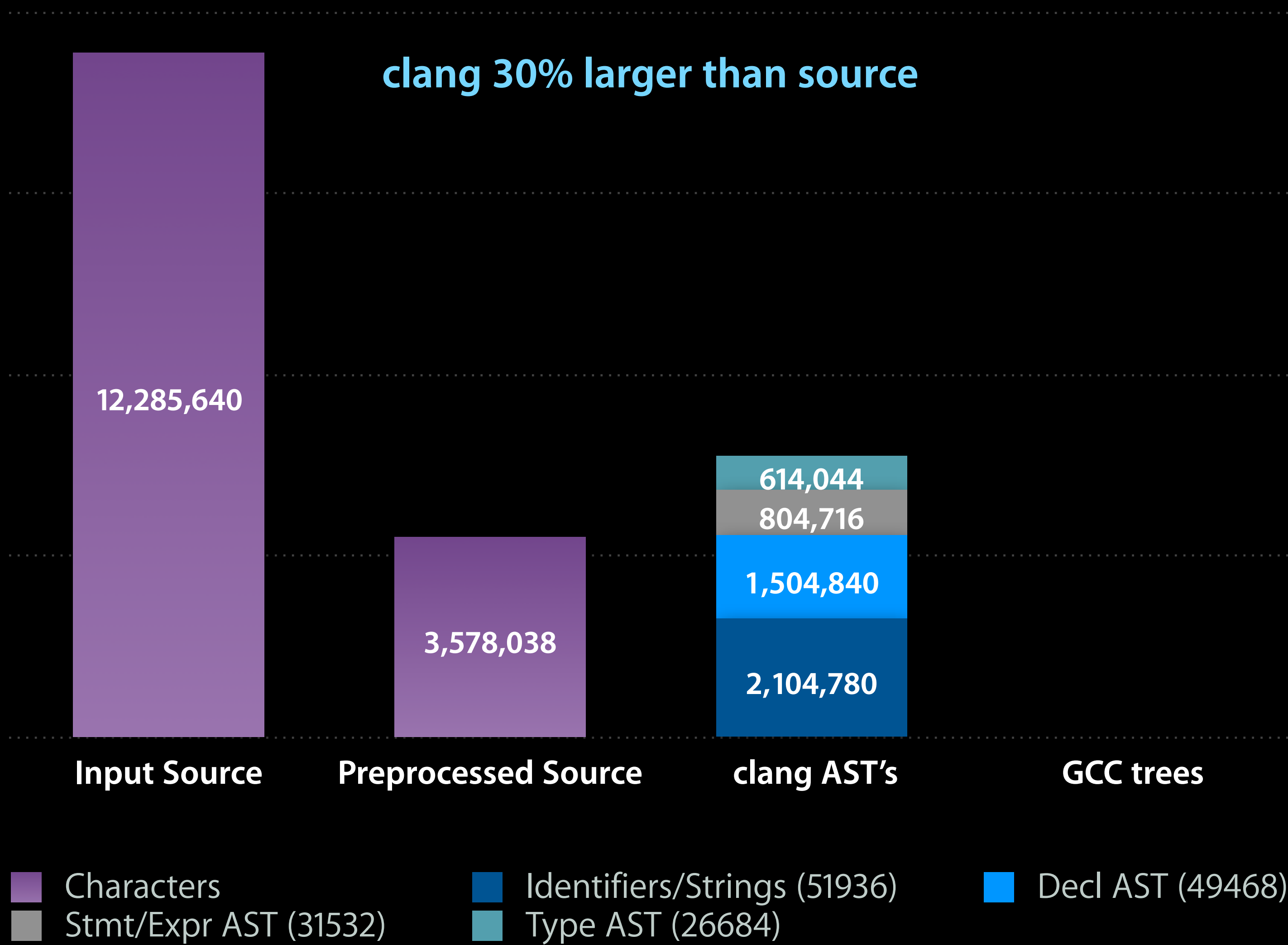
- Each pulls in carbon.h (C) or Cocoa.h (Objective C) among others
  - Standard interfaces to system frameworks

Problem: System headers are huge!

- Carbon.h contains:
  - 558 files
  - 12.3 megabytes!
  - 10,000 function declarations
  - 2000 structure definitions, 8000 fields
  - 3000 enum definitions, 20000 enum constants
  - 5000 typedefs
  - 2000 file scoped variables
  - 6000 macros

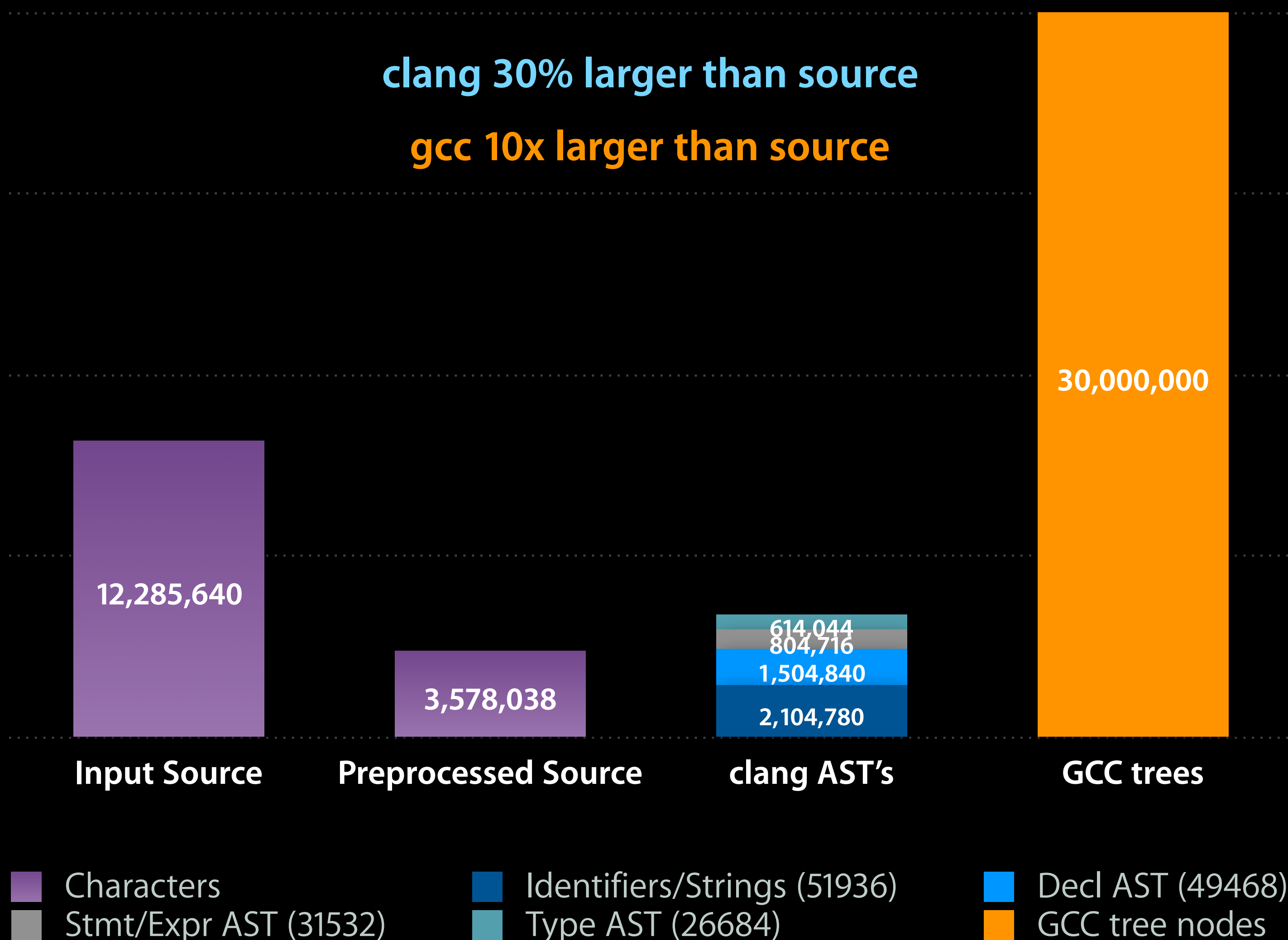
- Compile time is dominated by header preprocessing & analysis time:
  - Parser has to grok entire input
  - Optimizer and codegen only have to process code being used

Space



http://llvm.org/

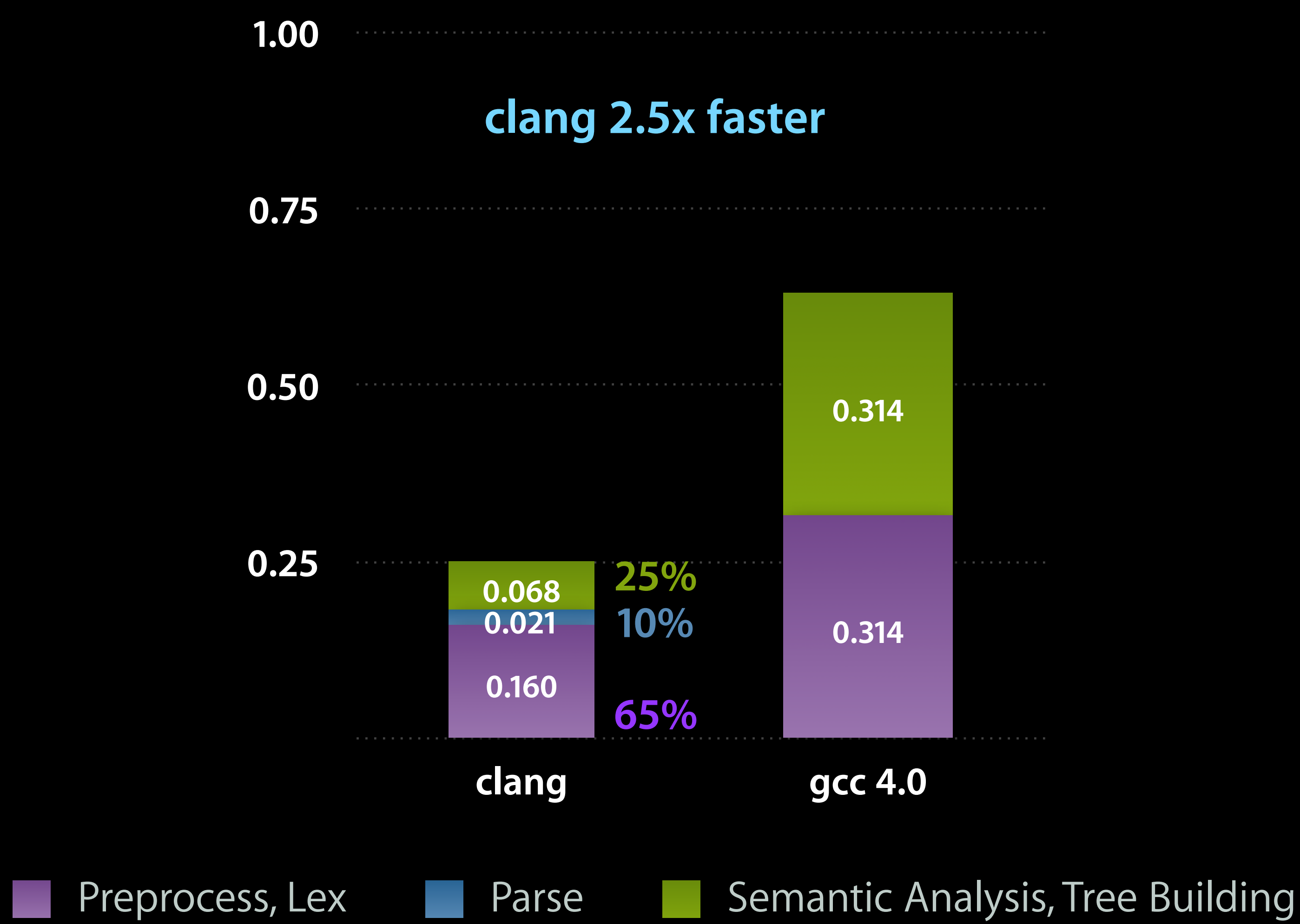
Space



http://llvm.org/

Time to parse carbon.h: -fsyntax-only

2.0 Ghz Intel Core Duo



http://llvm.org/

Changing the rules: 2.5x good, 10x better :-)

- Whole program ASTs (space efficient and easy to access)
  - Lazily [de]serialize them to/from disk or store in a server/IDE
  - ASTs are built as a side-effect of compiling
  - Use lessons learned from building and streaming LLVM IR
- ASTs are the intermediate form to enable many clients:
  - Symbol index, cross reference
  - Source code refactoring
  - Precompiled headers (smart caching)
  - Codegen to LLVM, in process (JIT for -O0?)
  - Debugger: use AST for types, decls, etc, reducing DWARF size
  - Syntax-aware highlighting (not regex'ing source) and editing
  - Who knows what else? Clearly many possibilities...

Next Question: What about parallel builds?

Distributed parallel builds with distcc

- distcc model:
  - Main build machine invokes distcc from makefile with high parallel build
  - Each distcc invocation preprocesses a source file and sends to slave
  - Slave compiles, optimizes, returns resultant .o file

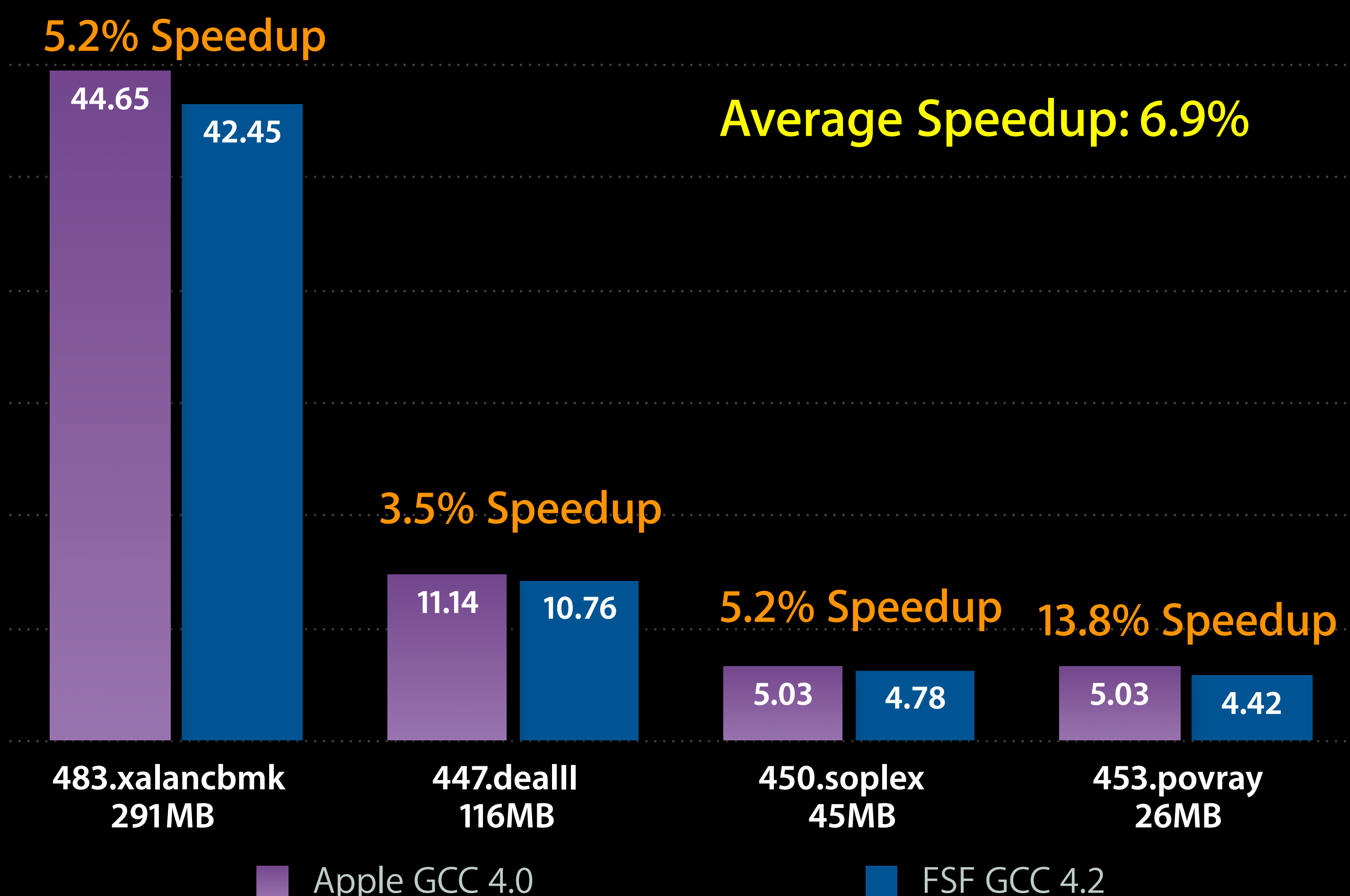
- Advantages:
  - Don't need to synch system, lib, app headers across machines
  - High optimization levels parallelize well

- Problem: Scalability limited by preprocessor!
  - "Performance tends to plateau between ten and twenty machines. This is consistent with measurements of the preprocessor using roughly 5-10% of the total CPU time: when twenty preprocessors and distcc clients are running, the client is completely saturated and cannot issue any more jobs."
  - From 'distcc, a fast free distributed compiler' by Martin Pool

What if you have huge clusters of machines laying around?

Preprocessor Speeds: GCC 4.0 vs 4.2

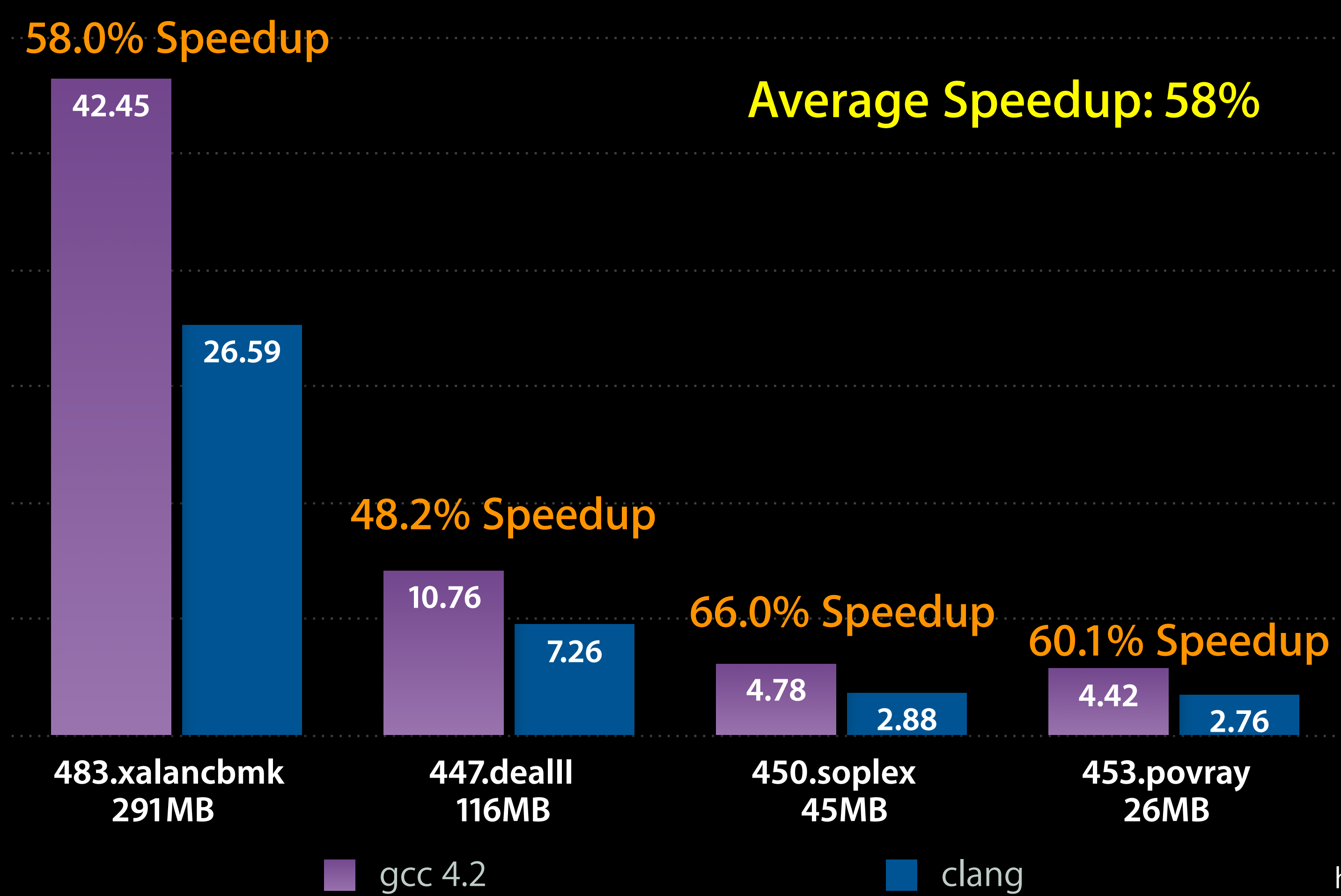
- Picked the biggest SPEC2006 benchmarks (by emitted .i file size)
  - For example, 483.xalancbmk is 291MB of preprocessed output!



http://llvm.org/

Preprocessor Speeds: GCC 4.2 vs clang

- Minimum of 3 runs, time on a 2.66Ghz Core2 Duo Mac Pro



http://llvm.org/

Changing the rules: Intelligent caching

- Observation: Huge amount of commonality across translation units
  - Header search paths are the same
  - Headers contents are common across all .cpp/.c files

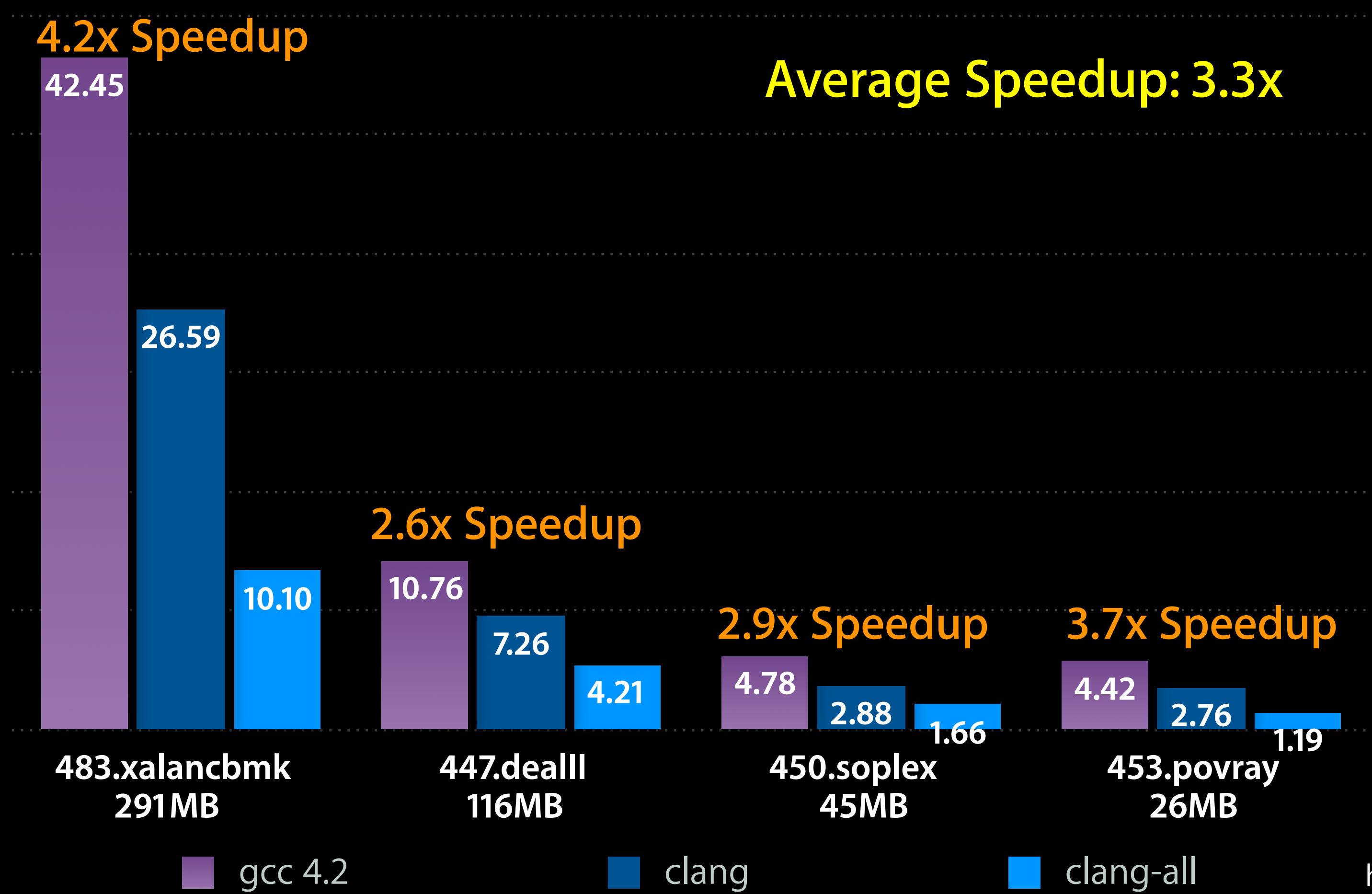
- Why not build a "distcc accelerator" tool?
  - clang is built as a reusable framework!

- Simple approach: cache the header lookup and filesystem objects
  - Model: Preprocess multiple files at once
  - Do cache basic filesystem accesses (mmaps, directory searches)
  - Don't cache preprocessor or lexer state - Possible for future work

- Note: implementation is not integrated with distcc
  - Patches welcome :-)

Preprocessor Speeds: GCC 4.2 vs clang-all

- 10x reduction in system time
  - 483.xalancbmk: clang: 9.85s, clang-all: 0.88s
- "-fdirectives-only": another 30%+ reduction in clang time (4x -> 12x)



http://llvm.org/

Conclusion: LLVM 2.0 and beyond!

- LLVM 2.0
  - Nearly "feature complete", many optimization and improvements
- LLVM 2.1, llvm-gcc 4.2
  - Coming in the next 3-6mo, new incremental improvements
- New C Frontend:
  - New library-based design, improves over GCC in many areas
  - Already has useful applications (3.3x speedup for distcc)
  - Still in early development

Questions?

http://llvm.org http://clang.llvm.org