A New Architecture for Building Software

Daniel Dunbar

Overview

- Compile time
- How software is built
- llbuild
- A new architecture

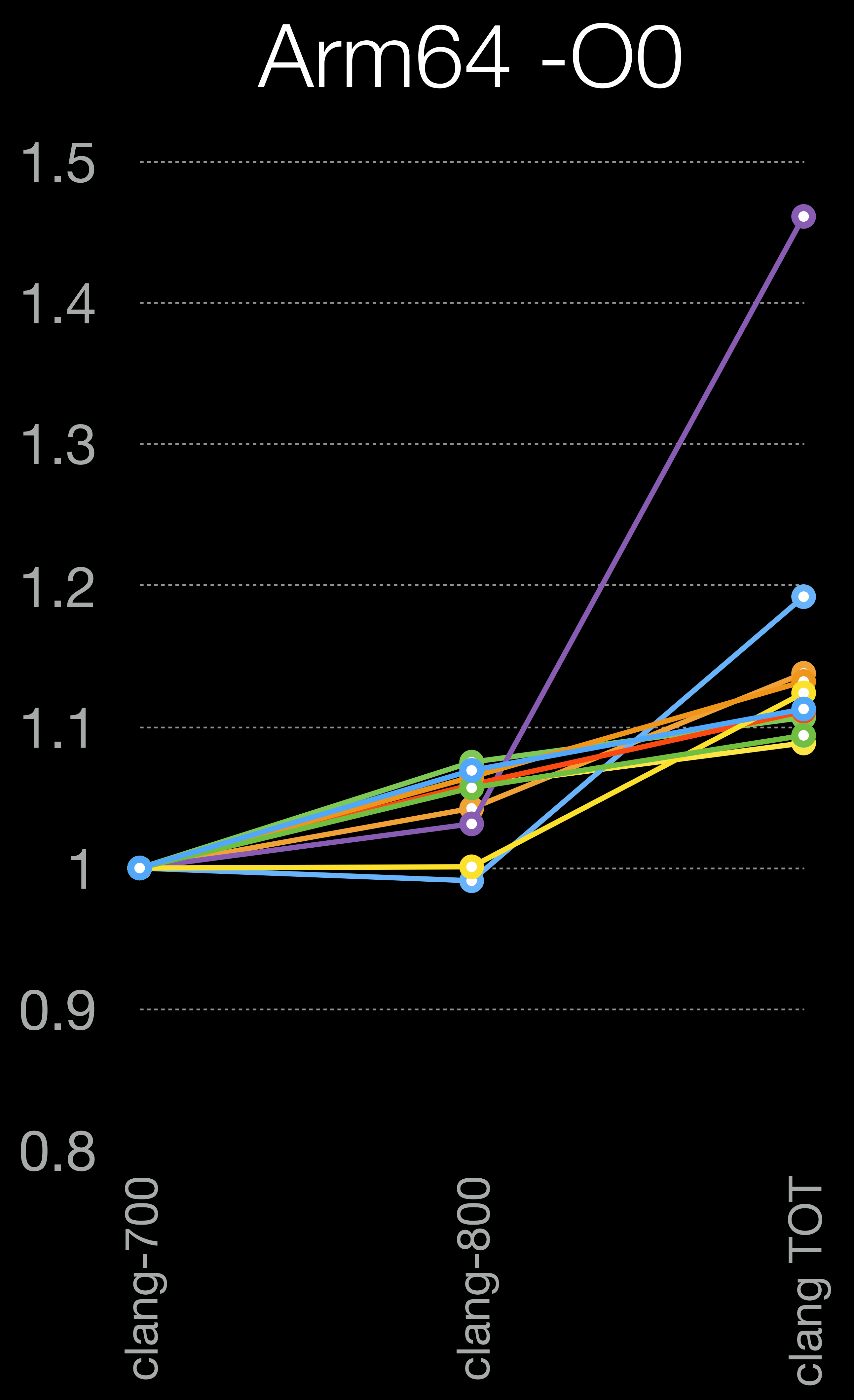
Compile Time

Clang & Compile Times

- Designed to be a fast compiler
  - Tuned lex & parse
  - Low-overhead -O0 path
  - Redesigned PCH implementation
  - Integrated assembler
- Very successful

Keeping Up With Compile Time

- Performance regresses
    - Features are added & tuning can break
    - Optimizing Clang is hard
- Occasional big wins
    - Bootstrap with link-time optimization
    - Enable order files
    - Modules
- Fewer architectural wins



Improving Compile Time

- Distributed compilation
- Fancy caching
  - Ideally distributed & shared
- Do less work ← (this talk)
  - … a lot less work
  - … ideally, O(N) less work

Clang calls `stat()` an average of **324 times** for each input file during the course of a Clang build.

What If I Told You…

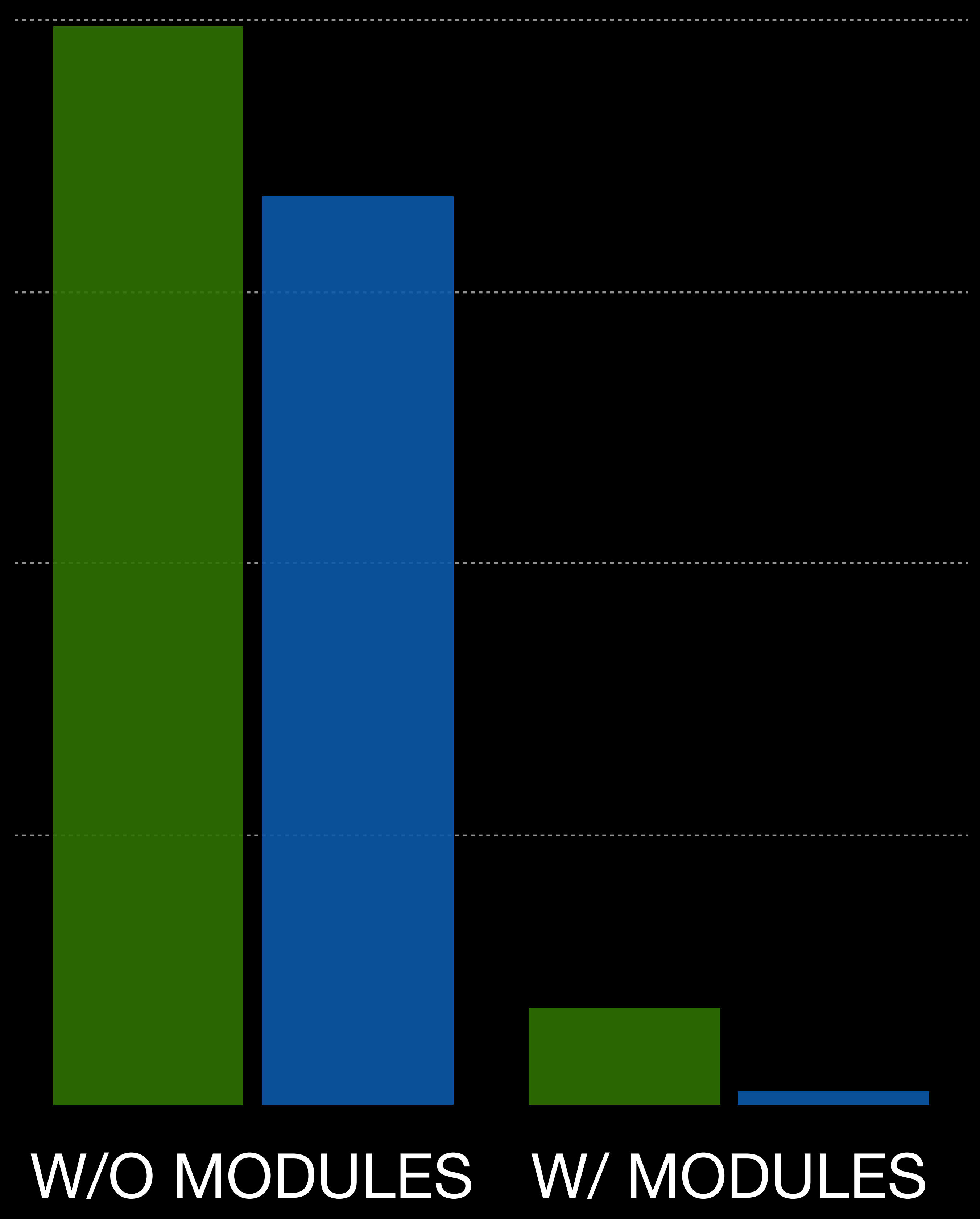
- 15% faster at type checking…
  - … without any work!

Frontend Source Sharing

- Clang frontend can process multiple TUs
  - Shares file & source managers
  - Works today
- … 85% faster with modules on

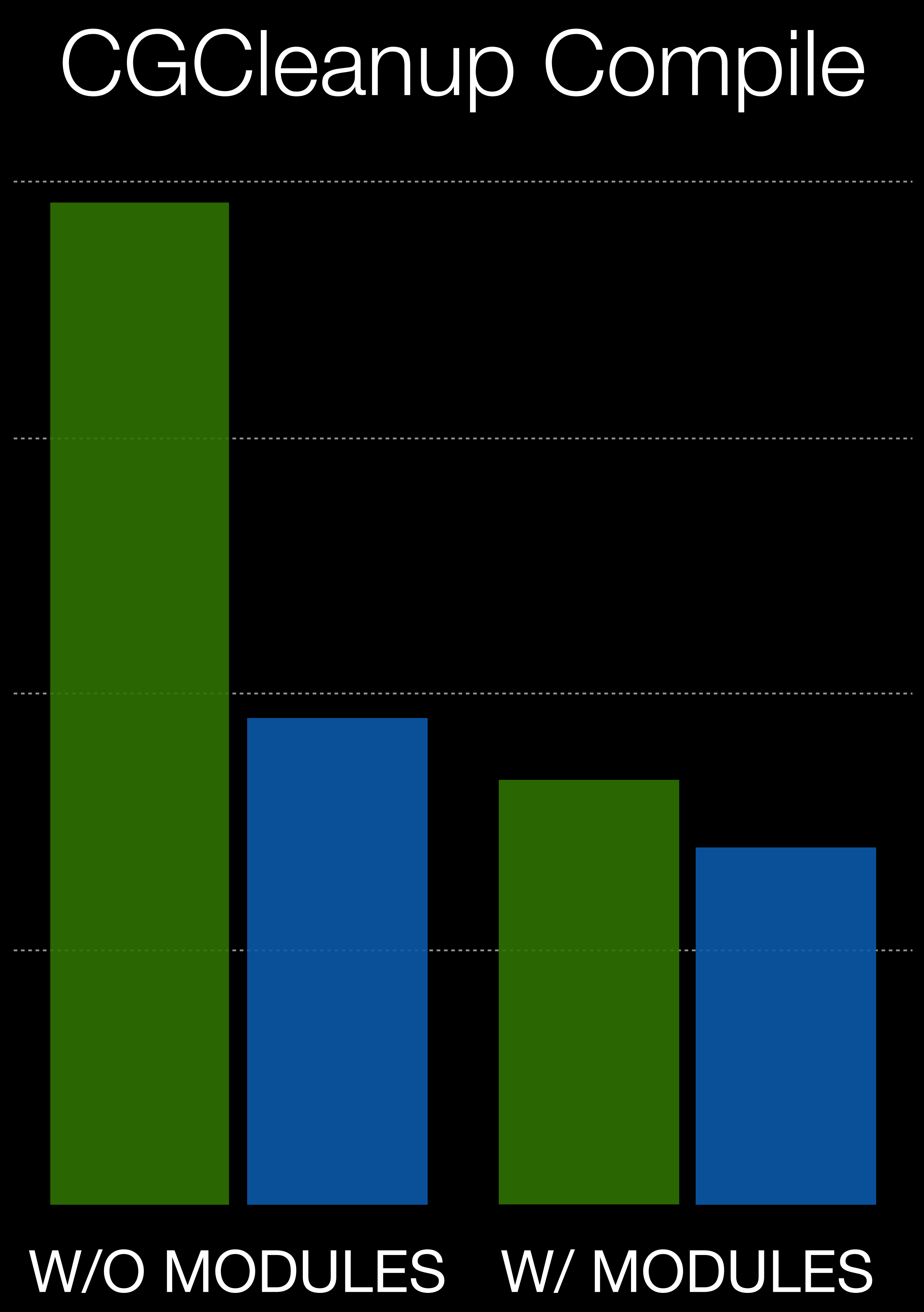
```
clang -fsyntax-only -x objective-c /dev/null \
           -Xclang t.m -Xclang t.m -Xclang t.m -Xclang t.m -Xclang t.m \
           -Xclang t.m -Xclang t.m -Xclang t.m -Xclang t.m -Xclang t.m
```

Cocoa Type Check



Precompiled Preamble

- Used in libclang for interactive editing
  - Automatically build PCH for “preamble”
  - Automatically reuse preamble when unchanged



Let’s Do It!

- Seems easy…
  - Shared compile flags? Reuse frontend!
  - Hotly edited file? Cache preamble!
- Uh oh!
  - No control over compiler invocation
  - Maybe if there was a compiler service…
- There must be a better way!

How Software Is Built

How Software Is Built

- Traditional UNIX compiler/build system model
  - Compiler runs as separate process
  - Primitive mechanisms for communicating dependencies
  - Fixed input/output pipeline defined by command line
- This is an API …
  - … and we haven’t changed it in decades
- We ❤️ breaking APIs

How Software Could Be Built

- Earlier examples are only the tip of the iceberg
  - Ad hoc lookup tables
  - Early exit via output signatures
  - Redundant template instantiations
- Need ability to evolve build system/compiler API
  - These changes need to be *easy*

What About The Module Cache?

- Clang's module cache solves this problem
  - Automatically builds modules when needed
  - Shares result across build
  - No build system changes required

An Nonexample: Module Cache

- Significant implementation complexity
    - File locking for coordination
    - Custom cache consistency management, few debugging tools
    - Custom cache eviction implementation (automatic pruning, tuning parameters)
    - Opaque to build system scheduler

Ideal Model for Building Software

- Support a flexible API between the compiler & build system
- Goals:
  - *Easy* to share redundant work
  - Compiler can optimize for *entire* build
  - Build system can optimize via rich compiler API
  - Consistent incremental builds & debuggable architecture

Ideal Model for Building Software

- Need ability to *integrate* build system and compiler
- Requires:

✅ Library-based compiler

❌ Extensible build system

❌ Compiler plugin

llbuild

Introducing llbuild

- llbuild is a new C++ library for building build systems
  - Uses LLVM ADT/Support & a library-based design philosophy
  - Open sourced as part of Swift project
  - Used in the Swift Package Manager
    - … and Swift Playgrounds
  - Contains a Ninja implementation

llbuild Goals

- Ignore build description / input language
- Focus on building a powerful engine
    - Support work being discovered on the fly
    - Scale to millions of tasks
    - Sophisticated scheduling
    - Powerful debugging tools
- Support a pluggable task API

llbuild Architecture

- Flexible underlying core engine
  - Library for persistent, incremental computation
  - Heavily inspired by a Haskell build system called Shake
  - Low-level
    - Inputs & outputs are byte-strings
    - Functions are abstract
    - Use C++ API between tasks
- Higher-level build systems are built on the core

llbuild Engine

- Minimal, functional model
  - **Key**: Unambiguous name for a computation
  - **Value**: The result of a computation
  - **Rule**: How to produce a **Value** for a **Key**
  - **Task**: A running instance of a **Rule**
    - A task can request other input **Keys** as part of its work

llbuild	make/ninja
Key	/a/b.o
Value	stat("/a/b.o")
Rule	/a/b.o: /a/b.c
Task	fork/exec

An Example: Recursive Functions

- Core engine can be used directly for general computation
- Recursive functions form a natural graph
  - Each result depends on the recursive inputs
- Let’s build Ackermann!

```
auto ack(int m, int n) -> int {
  if (m == 0) {
    return n + 1;
  } else if (n == 0) {
    return ack(m - 1, 1);
  } else {
    return ack(m - 1, ack(m, n - 1));
  };
}
```

“Building” Ackermann

- Computing Ackermann with llbuild:
  - Encode function invocation as **key**: `ack(3,14)`
  - Encode integer result as **value**
  - **Rules** map keys like `ack(3,14)` to a task
  - **Tasks** implement the Ackermann function

Ackermann: Keys

```
#include "llbuild/Core/BuildEngine.h"

using namespace llbuild;

/// Key representation used in Ackermann build.
struct AckermannKey {
  /// The Ackermann number this key represents.
  int m, n;

  /// Create a key representing the given Ackermann number.
  AckermannKey(int m, int n) : m(m), n(n) {}

  /// Create an Ackermann key from the encoded representation.
  AckermannKey(const core::KeyType& key) { … }

  /// Convert an Ackermann key to its encoded representation.
  operator core::KeyType() const { … }
};
```

Ackermann: Values

```
/// Value representation used in Ackermann build.
struct AckermannValue {
  /// The wrapped value.
  int value;

  /// Create a value from an integer.
  AckermannValue(int value) : value(value) { }

  /// Create a value from the encoded representation.
  AckermannValue(const core::ValueType& value) : value(intFromValue(value)) { }

  /// Convert a value to its encoded representation.
  operator core::ValueType() const { … }
};
```

Ackermann: Rules

```
/// An Ackermann delegate which dynamically constructs rules like "ack(m,n)".
class AckermannDelegate : public core::BuildEngineDelegate {
public:
  /// Get the rule to use for the given Key.
  virtual core::Rule lookupRule(const core::KeyType& keyData) override {
    auto key = AckermannKey(keyData);
    return core::Rule{key, [key] (core::BuildEngine& engine) {
        return new AckermannTask(engine, key.m, key.n); } };
  }

  /// Called when a cycle is detected by the build engine and it cannot make
  /// forward progress.
  virtual void cycleDetected(const std::vector<core::Rule*>& items) override { … }
};
```

Ackermann: Tasks

```
/// Compute the result for an individual Ackermann number.
struct AckermannTask : core::Task {
  int m, n;
  AckermannValue recursiveResultA, recursiveResultB;

  AckermannTask(core::BuildEngine& engine, int m, int n) : m(m), n(n) {
    engine.registerTask(this);
  }

  /// Called when the task is started.
  virtual void start(…) override { … }

  /// Called when a task's requested input is available.
  virtual void provideValue(…) override { … }

  /// Called when all inputs are available.
  virtual void inputsAvailable(…) override { … }
};
```

Ackermann: Tasks

```
/// Compute the result for an individual Ackermann number.
struct AckermannTask : core::Task {
  …

  /// Called when the task is started.
  virtual void start(core::BuildEngine& engine) override {
    // Request the first recursive result, if necessary.
    if (m == 0) {
      ;
    } else if (n == 0) {
      engine.taskNeedsInput(this, AckermannKey(m-1, 1), 0);
    } else {
      engine.taskNeedsInput(this, AckermannKey(m, n-1), 0);
    }
  }

  …
}
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m\text{-}1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m\text{-}1, A(m\text{-}1, n\text{-}1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Ackermann: Tasks

```
/// Compute the result for an individual Ackermann number.
struct AckermannTask : core::Task {
  …
  /// Called when a task's requested input is available.
  virtual void provideValue(core::BuildEngine& engine, uintptr_t inputID,
                            const core::ValueType& value) override {
    if (inputID == 0) {
      recursiveResultA = value;

      // Request the second recursive result, if needed.
      if (m > 0 && n > 0) {
        engine.taskNeedsInput(this, AckermannKey(m-1, recursiveResultA), 1);
      }
    } else {
      recursiveResultB = value;
    }
  }
  …
}
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m\text{-}1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m\text{-}1, A(m\text{-}1, n\text{-}1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Ackermann: Tasks

```
/// Compute the result for an individual Ackermann number.
struct AckermannTask : core::Task {
  …
  /// Called when all inputs are available.
  virtual void inputsAvailable(core::BuildEngine& engine) override {
    if (m == 0) {
      engine.taskIsComplete(this, AckermannValue(n + 1));
      return;
    }

    if (n == 0) {
      engine.taskIsComplete(this, recursiveResultA);
      return;
    }

    engine.taskIsComplete(this, recursiveResultB);
  }
};
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m-1, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

`

```
/// Compute an Ackermann number using llbuild.
void runAckermannBuild(int m, int n) {
  /// Create the build engine delegate.
  AckermannDelegate delegate;

  /// Create the engine.
  core::BuildEngine engine(delegate);

  /// Build and report the result.
  auto result = AckermannValue(engine.build(AckermannKey(m, n)));
  llvm::errs() << "ack(" << m << ", " << n << ") = " << result << "\n";
}
```

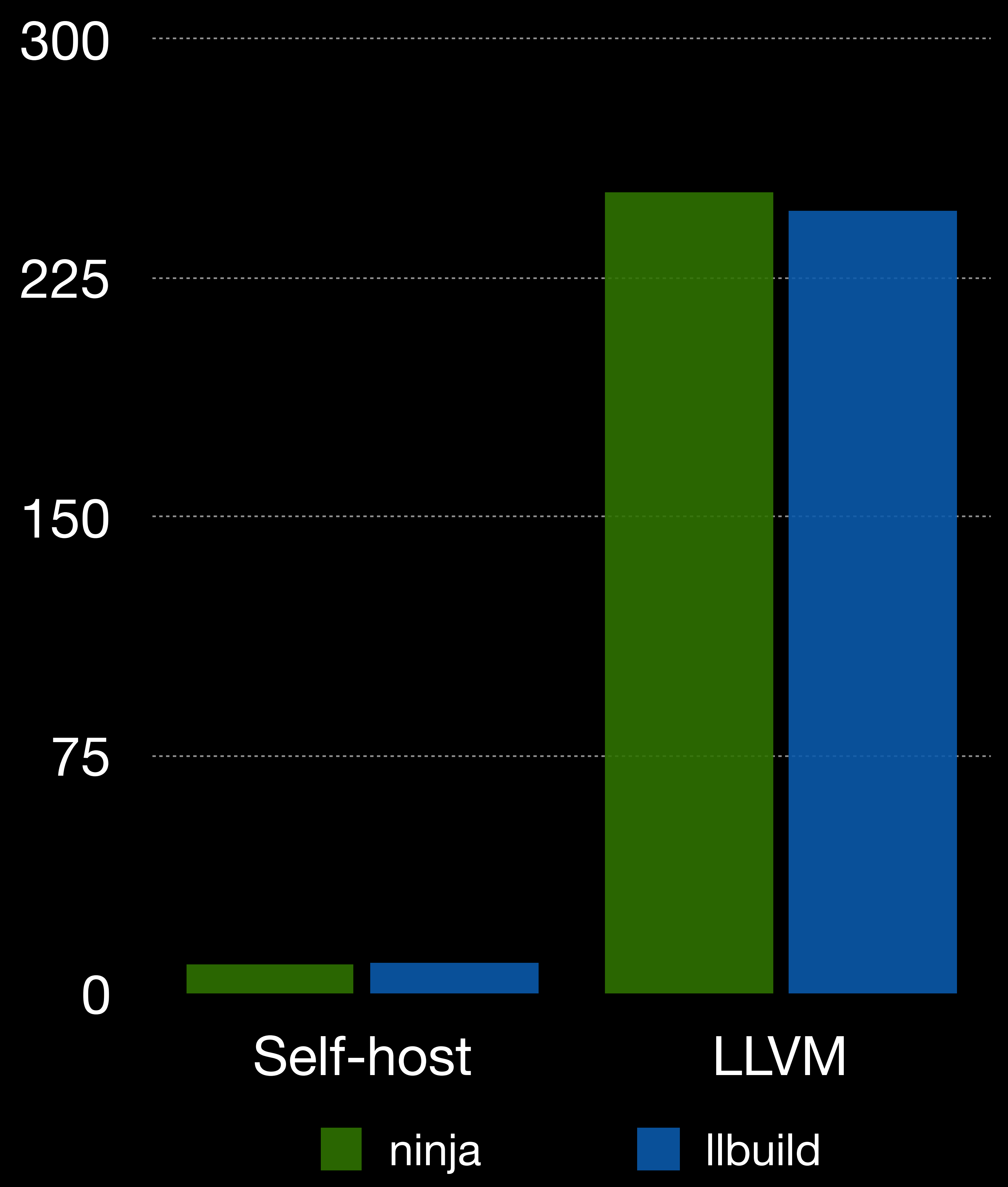
```
$ time llbuild buildengine ack 3 14
ack(3, 14) = 131069
... computed using 327685 rules

real 0m1.056s
user 0m0.925s
sys  0m0.116s
```

42 times more rules than LLVM + Clang

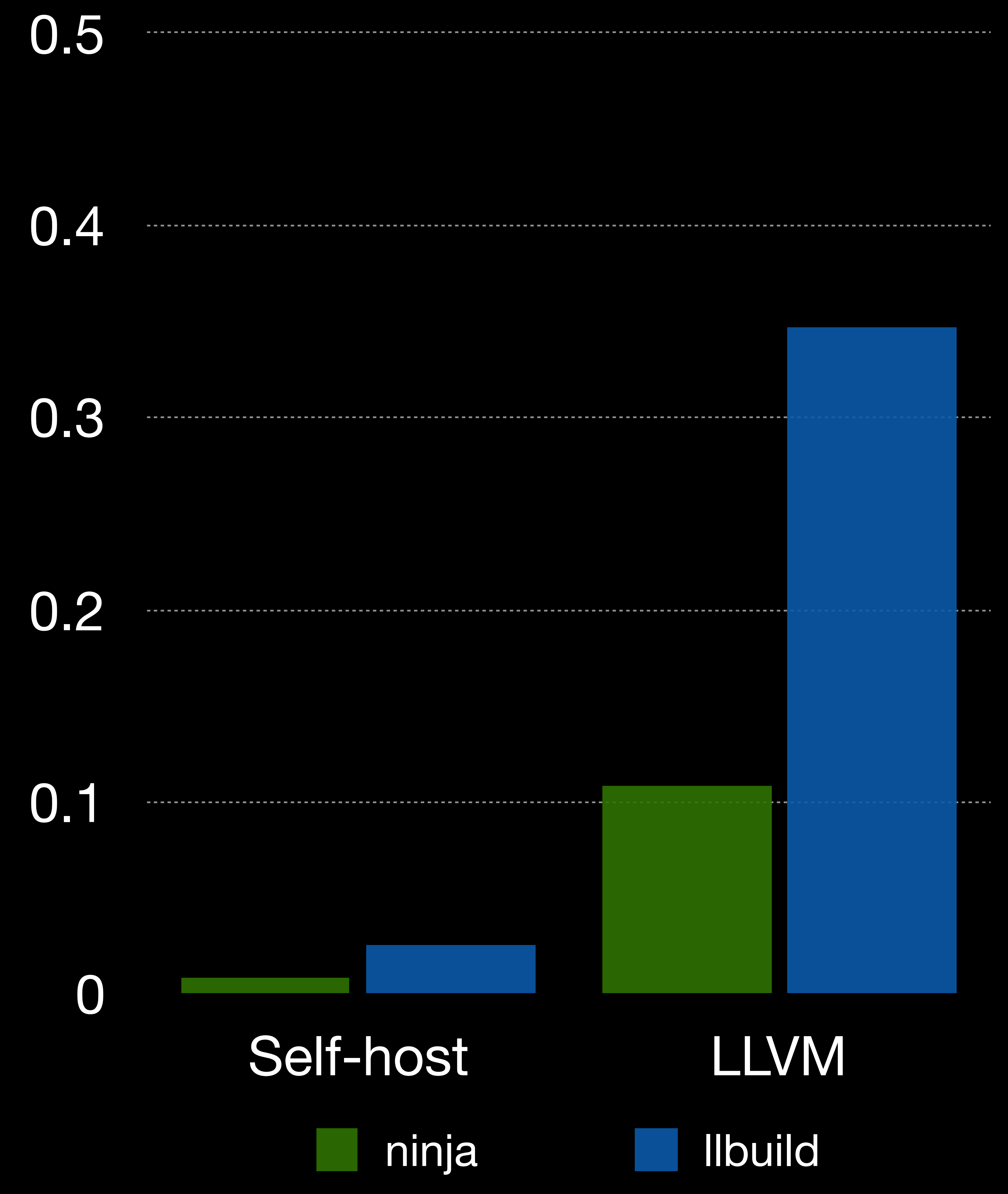
llbuild Performance

- Wall times for full parallel build
- Two test projects:
  - llbuild self-host
  - LLVM (x86 only)



llbuild Performance

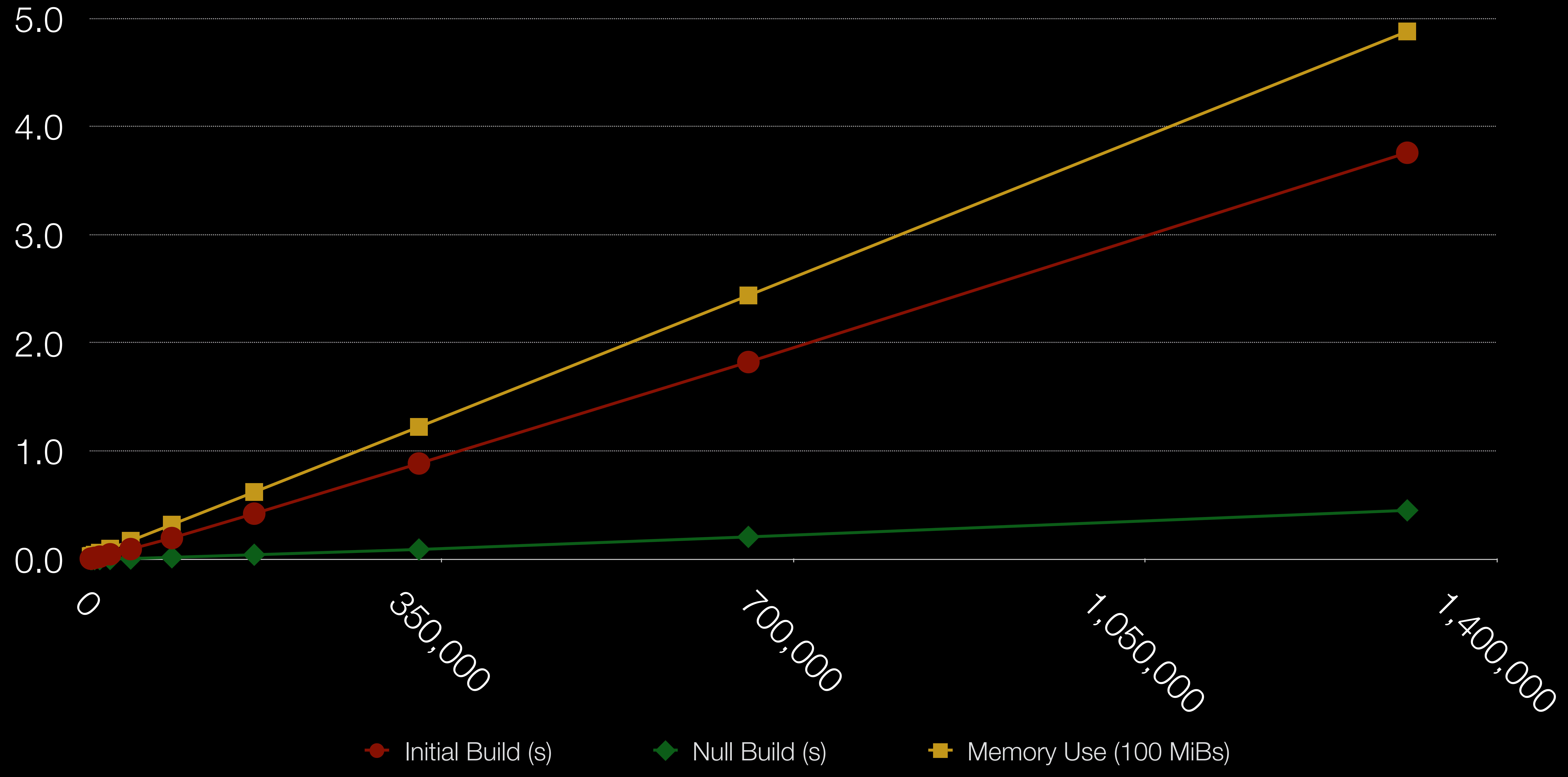
- Wall times for null build
- Two test projects:
  - llbuild itself
  - LLVM (x86 only)



llbuild Scalability

- Designed to scale to large graphs
- Validate by looking for linear performance vs size
- Experiments done using the Ackermann function

llbuild Scalability



A New Architecture

A New Architecture

- Requires:

  ✅ Library-based compiler

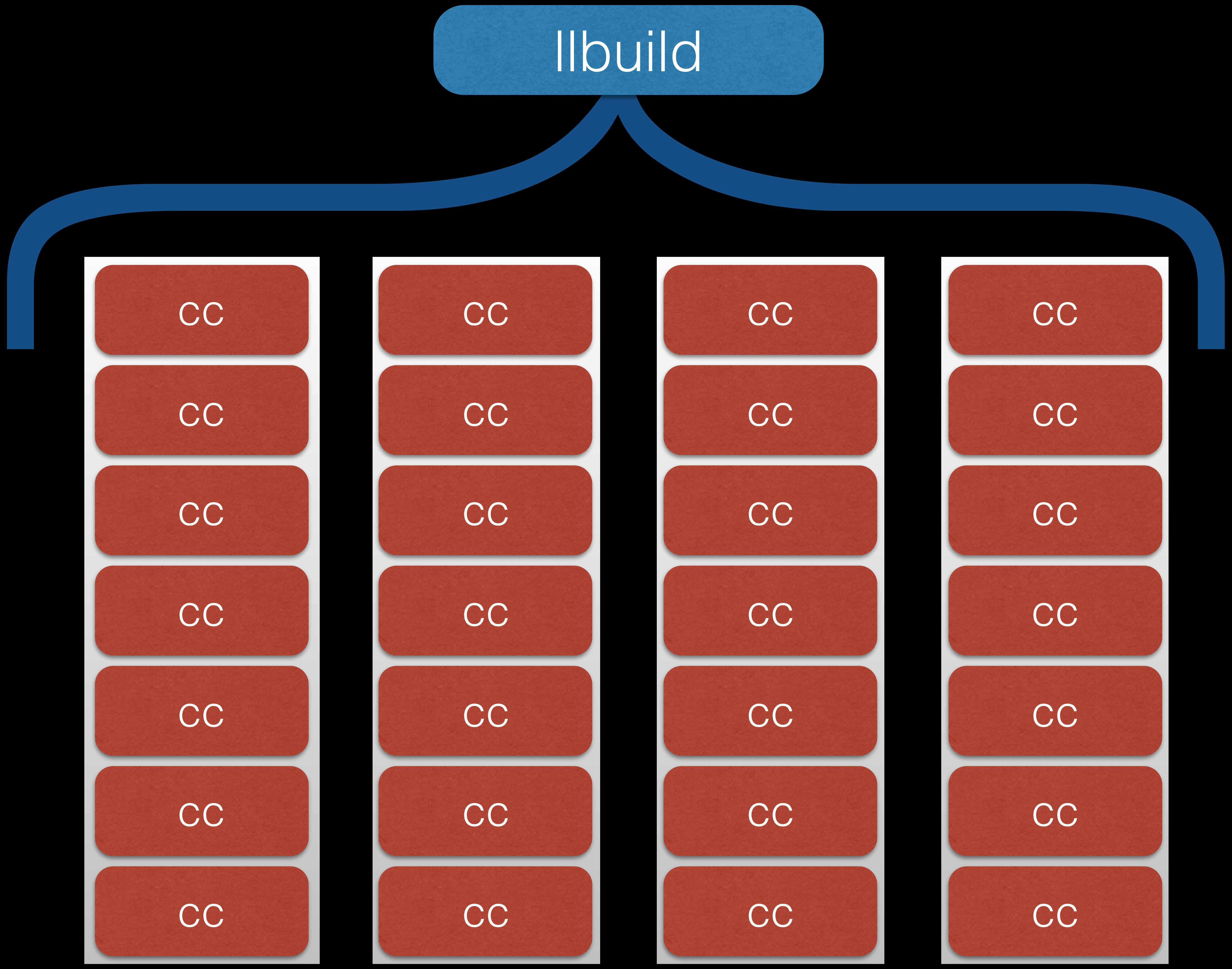
  ✅ Extensible build system

  ❌ Compiler plugin

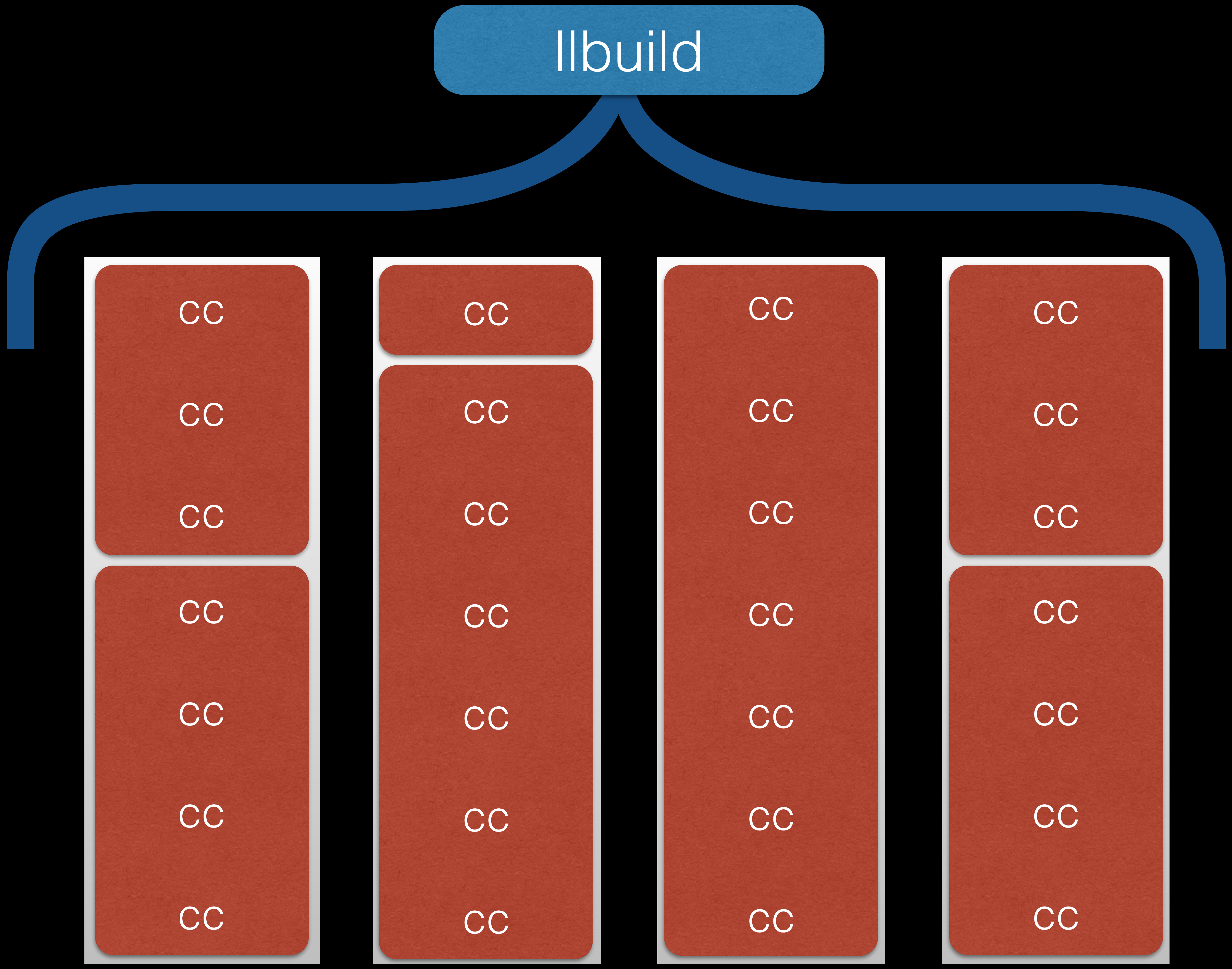
Clang Compiler Plugin

- A straw man proposal
  - Focus on easiest path to vet concept
  - Add a minimal new protocol for controllable compiler subprocess
    - Use JSON (etc.) to send & receive commands
  - Share subprocesses when available
    - Dispatch individual compile requests as they arrive
    - Restart subprocess on crashes, etc.

Current Model



Proposed Shared Frontend



Proposed Shared Frontend

- Enables file & source manager sharing
- Amortizes module validation time
- Avoids need to make full compiler thread safe
- Gives us a new API to break!

Summary

- The current compiler / build system split is a *legacy API*
    - Potentially large compile time wins by evolving
- llbuild: https://github.com/apple/swift-llbuild
    - As Ninja: `llbuild ninja build` (or `ln -s llbuild ninja`)
    - Docs: https://github.com/apple/swift-llbuild/tree/master/docs
    - Ackermann: `lib/Commands/BuildEngineCommand.cpp`

This Slide Intentionally Left Blank