clang-scan-deps

Fast Dependency Scanning For Explicit Modules

Alex Lorenz, Michael Spencer, Apple

LLVM Developers' Meeting, Brussels, Belgium, April 2019

Clang Modules

Dependency Scanning

Fast Dependency Scanning

Dependency Extraction

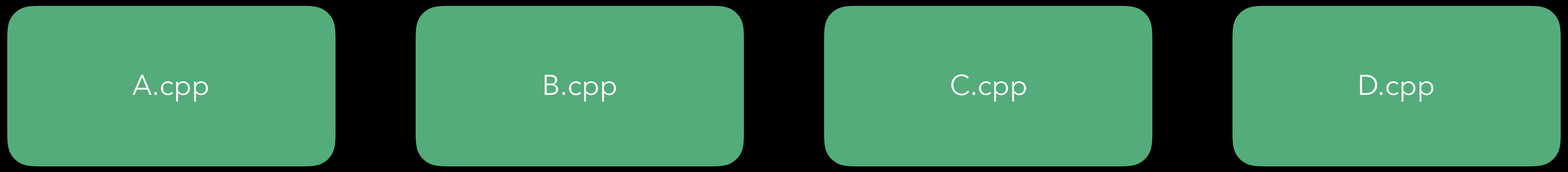
Future Work

Clang Modules

- Replace the textual preprocessor inclusions with an import of an AST
- Widely used in SDKs shipped with Xcode
  - Implicit modules: Clang builds modules as they’re included
  - Users don’t have to specify modular dependencies 🙂
  - Requires a build system in the compiler 😫

Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



A.cpp

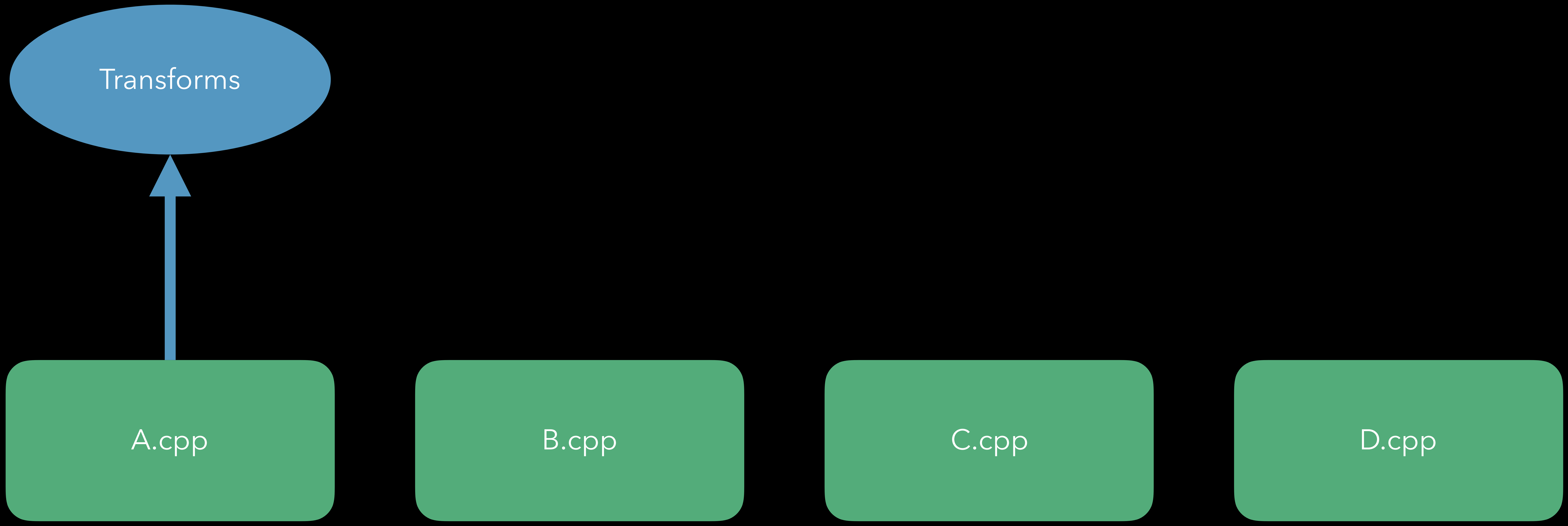
B.cpp

C.cpp

D.cpp

Implicit Modules

● Compiler Discovered ● Build System Known



Transforms

A.cpp

B.cpp

C.cpp

D.cpp

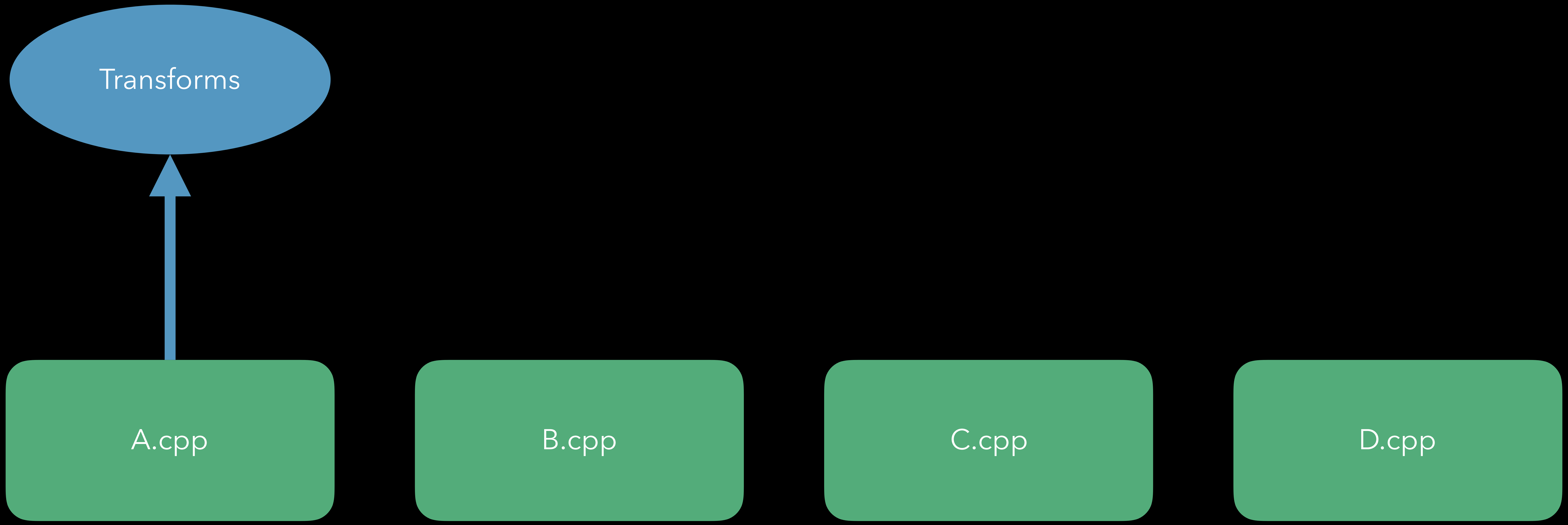
Implicit Modules

Module Maps

```
module LLVM_Transforms {
  requires cplusplus
  umbrella "Transforms"
  module * { export * }
}
```

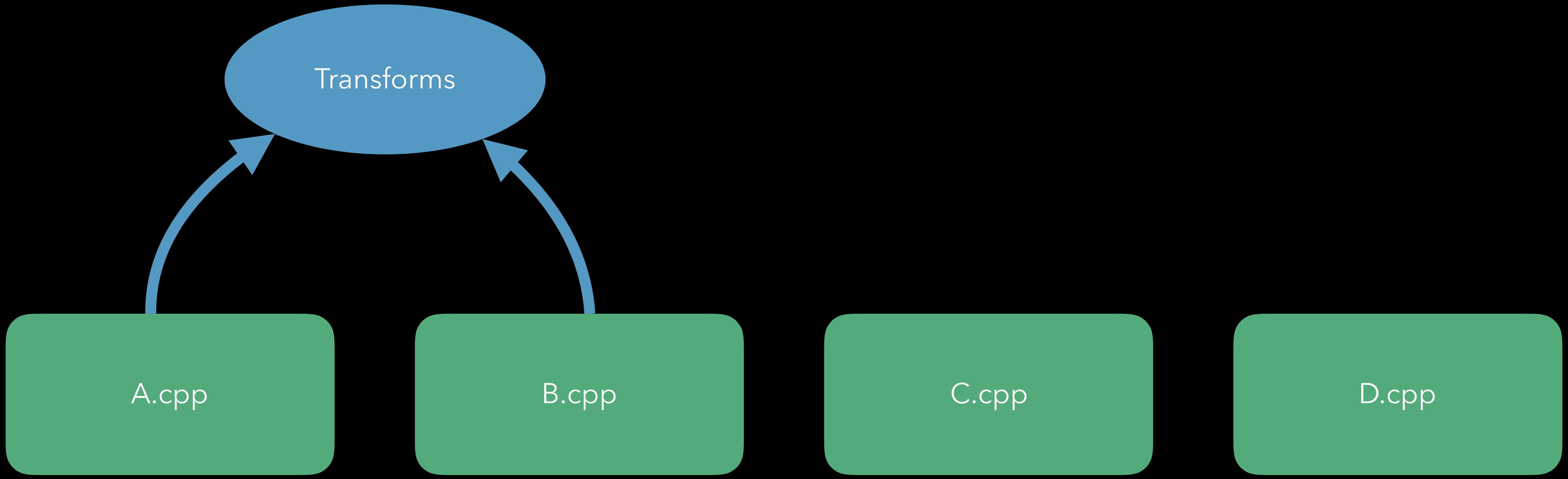
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



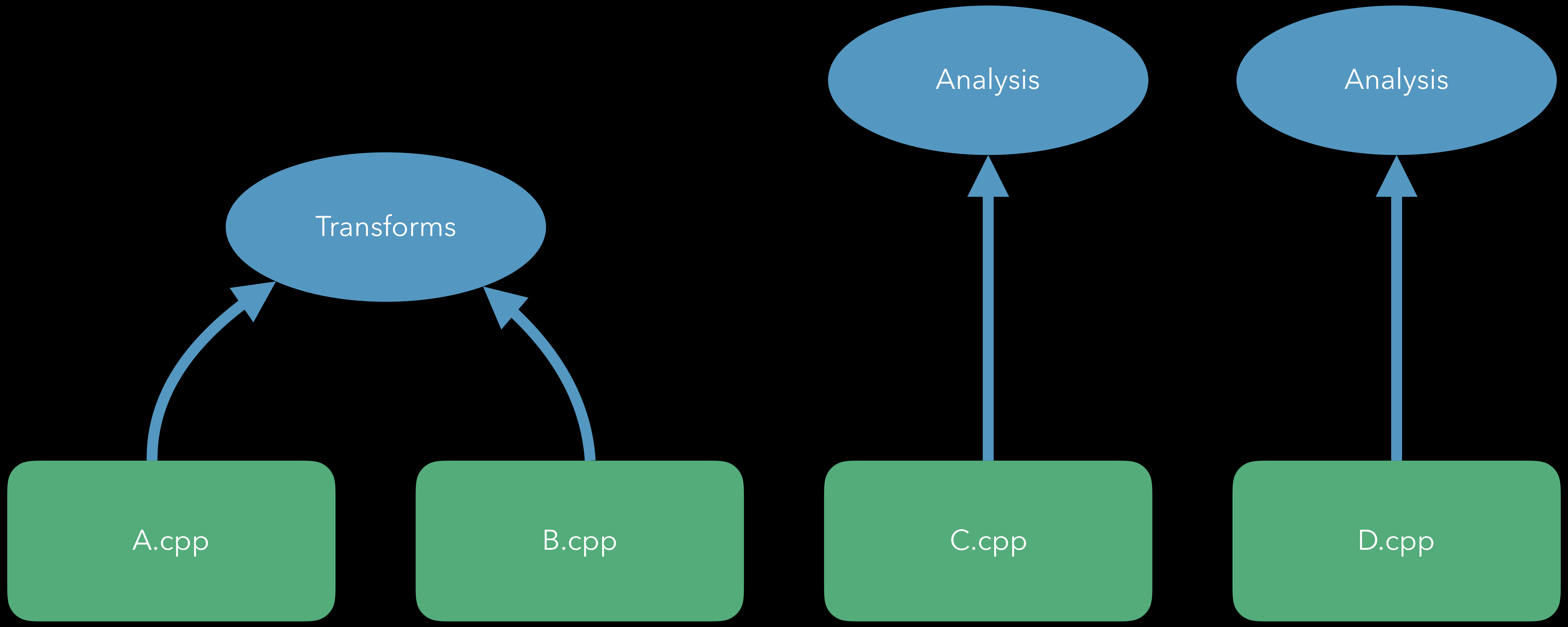
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



Implicit Modules

● Compiler Discovered ● Build System Known



Transcforms

Analysis

Analysis

A.cpp

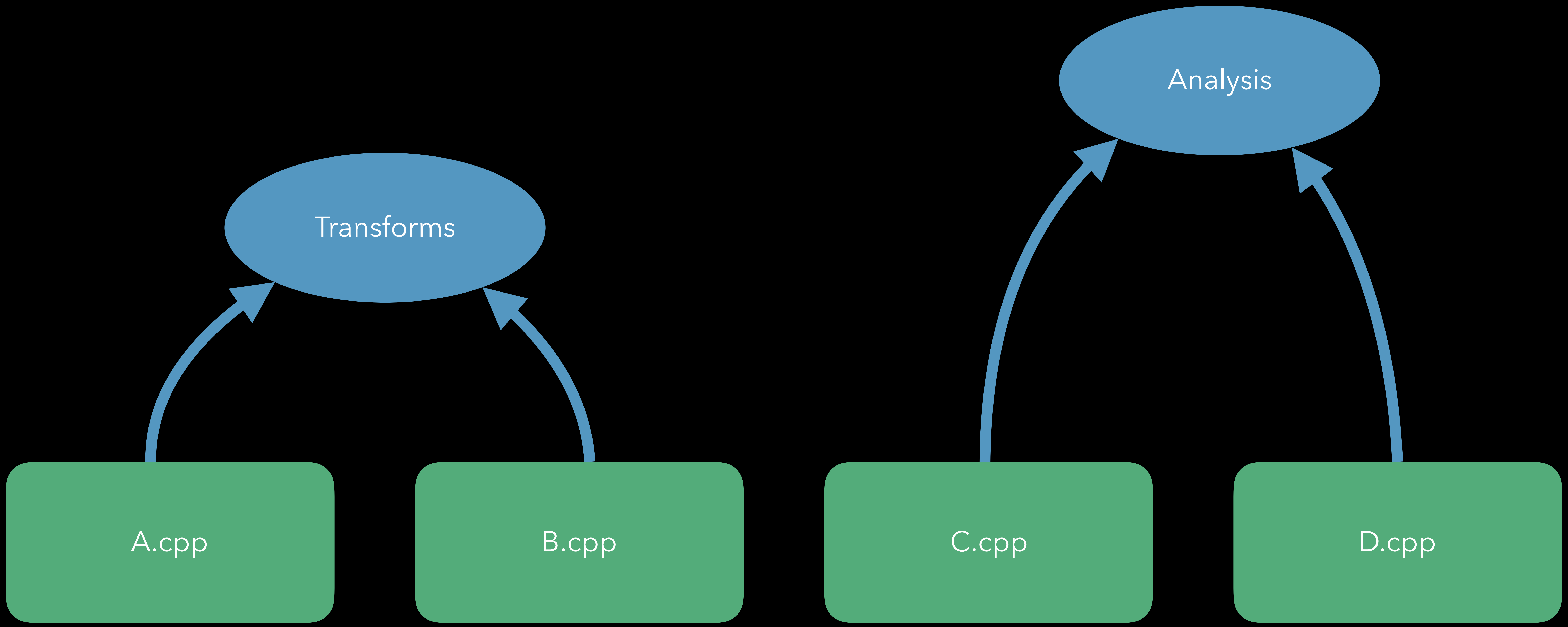
B.cpp

C.cpp

D.cpp

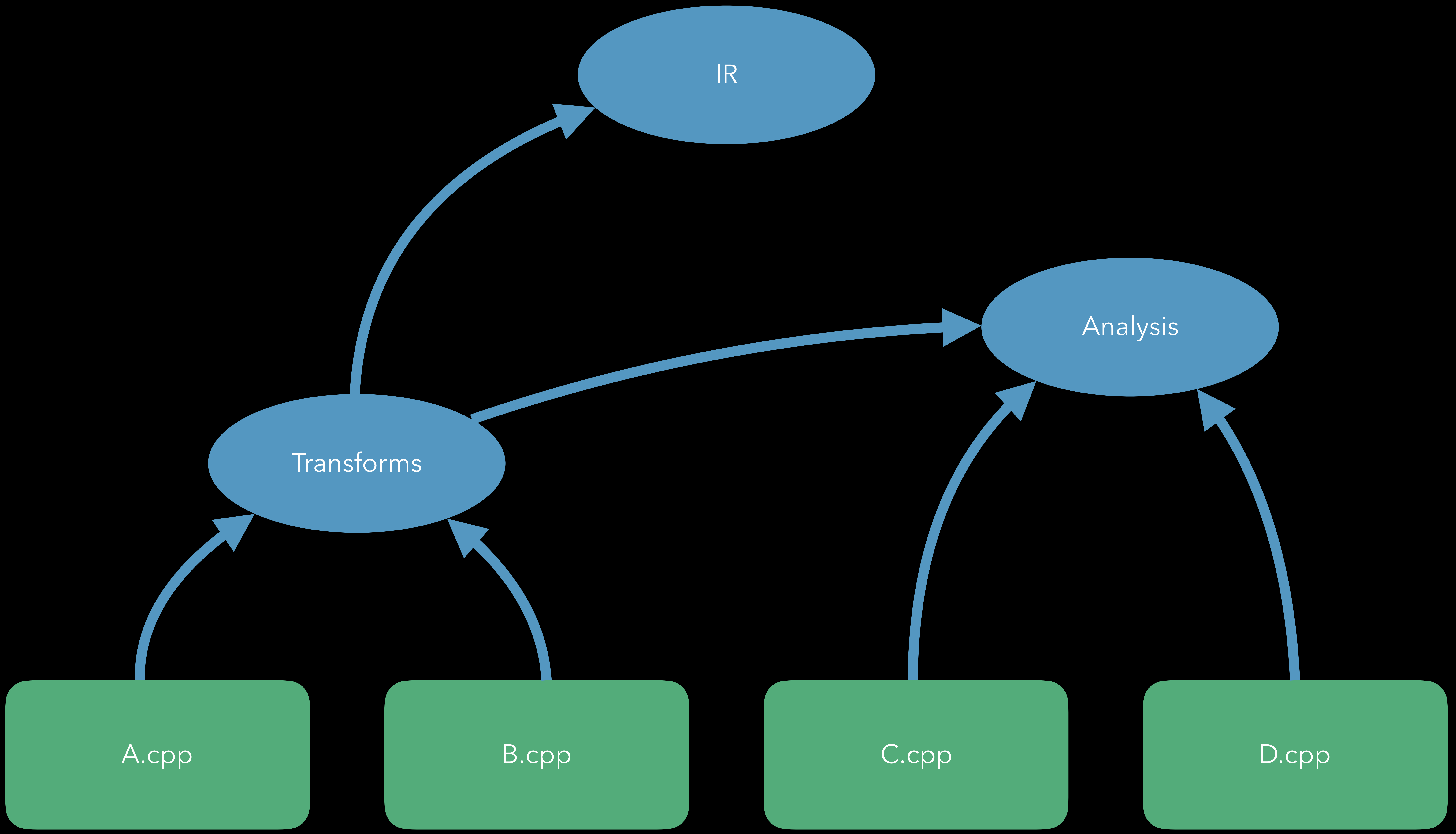
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



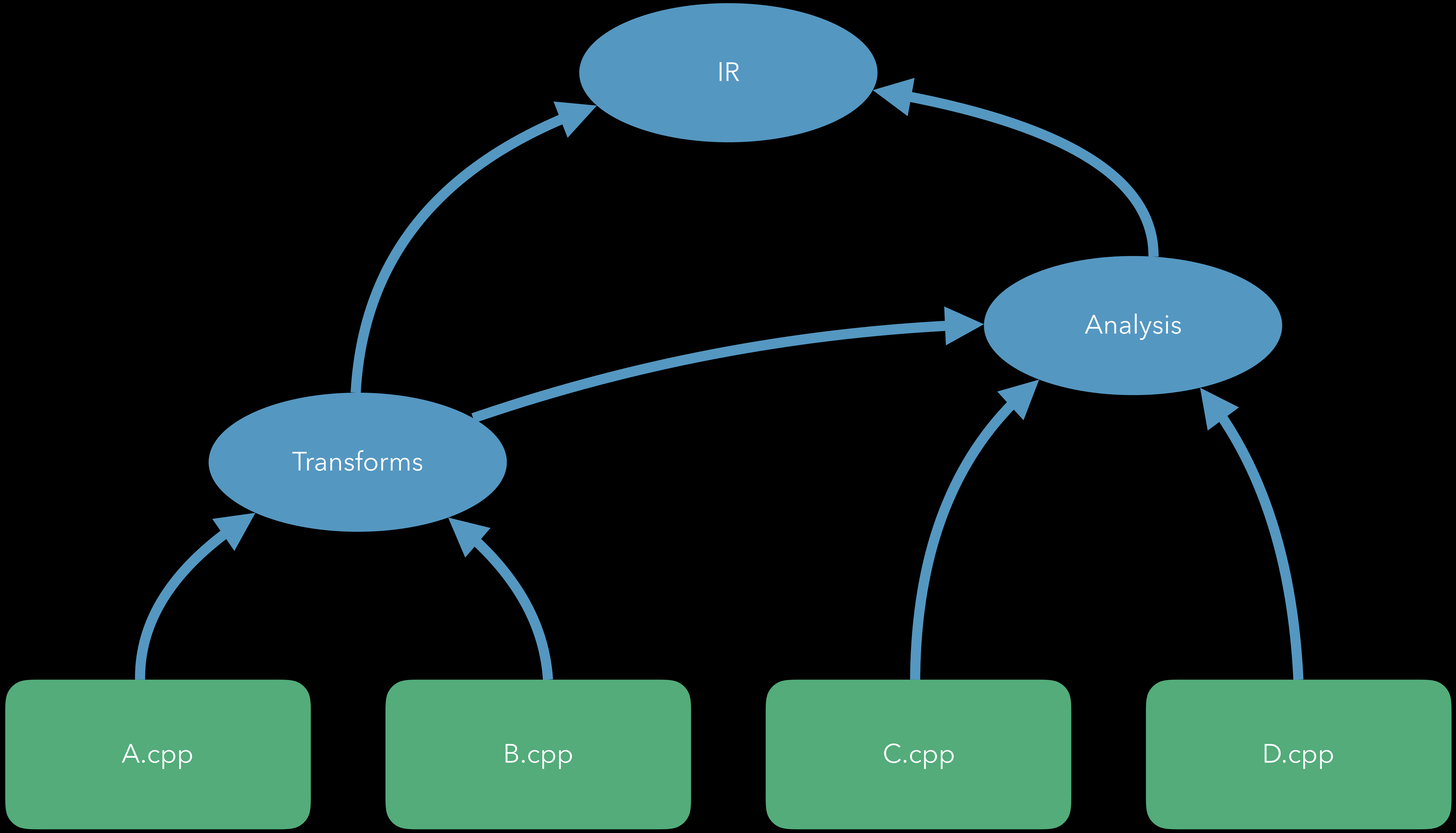
Implicit Modules

● Compiler Discovered ● Build System Known



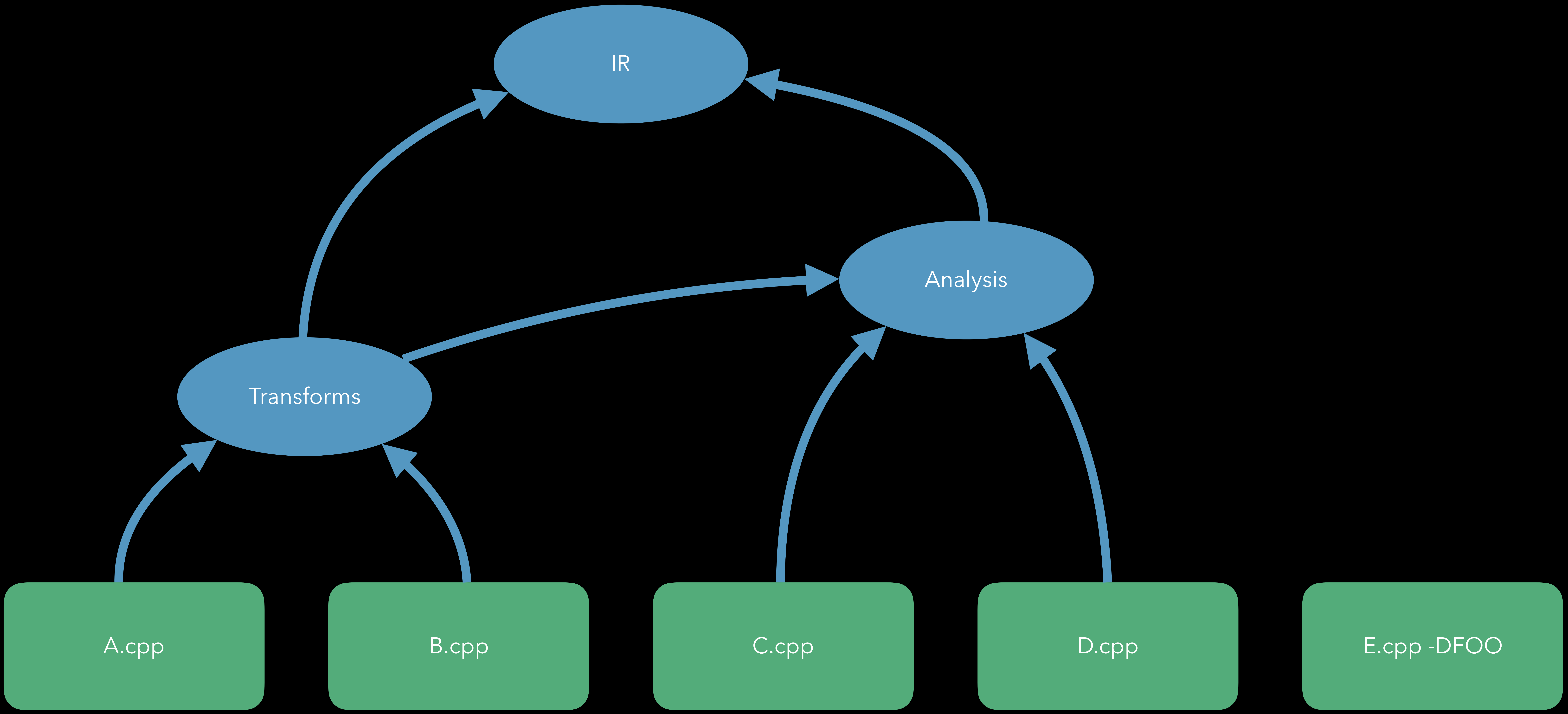
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



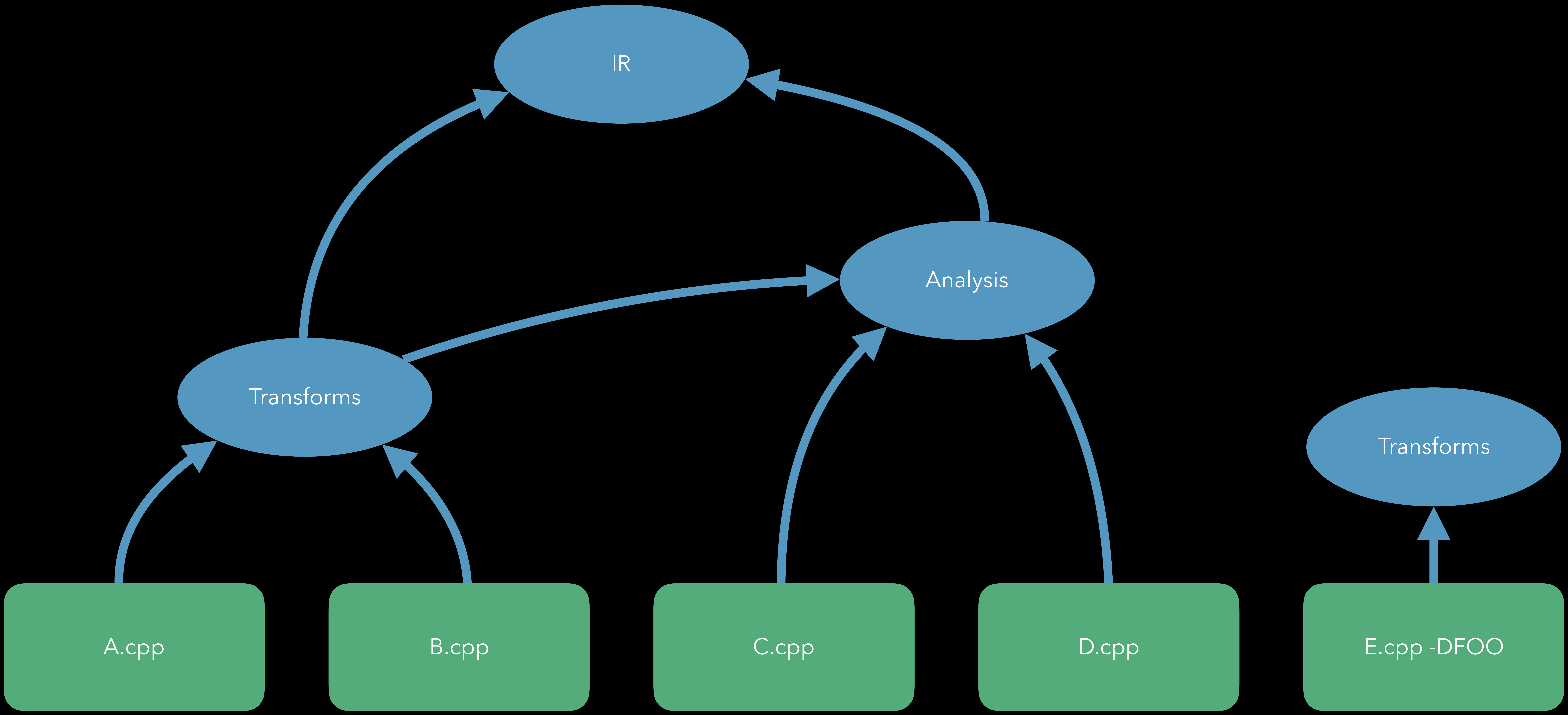
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



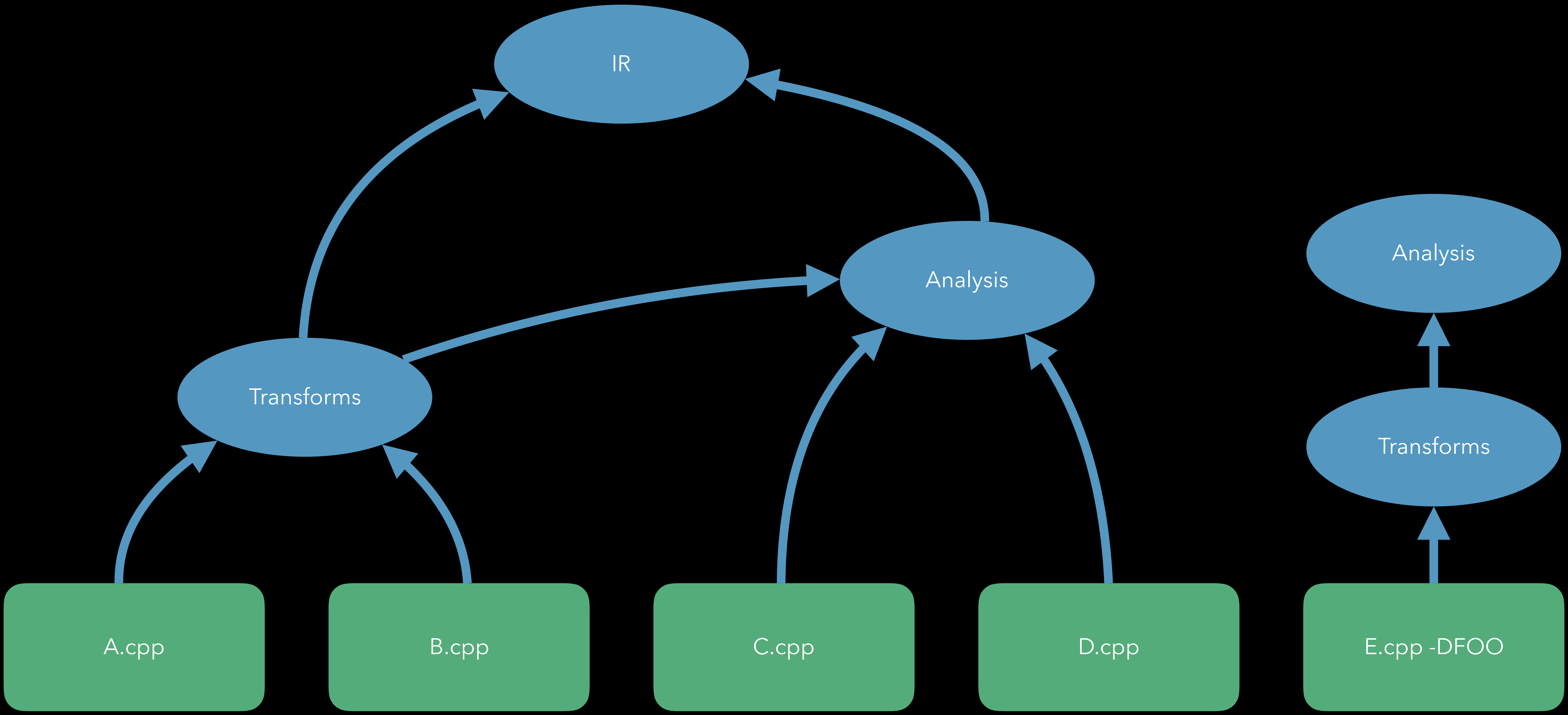
Implicit Modules

● Compiler Discovered ● Build System Known



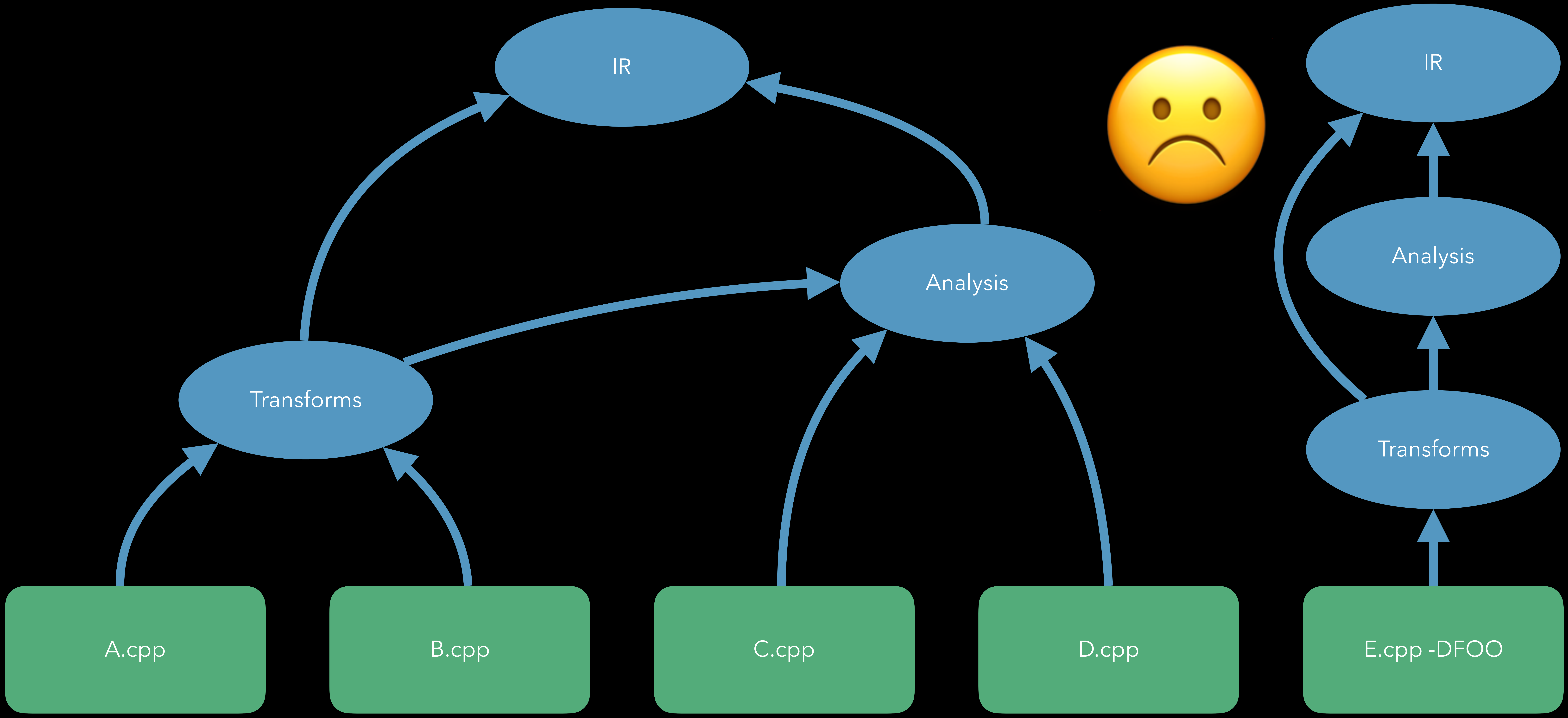
Implicit Modules

● Compiler Discovered ● Build System Known



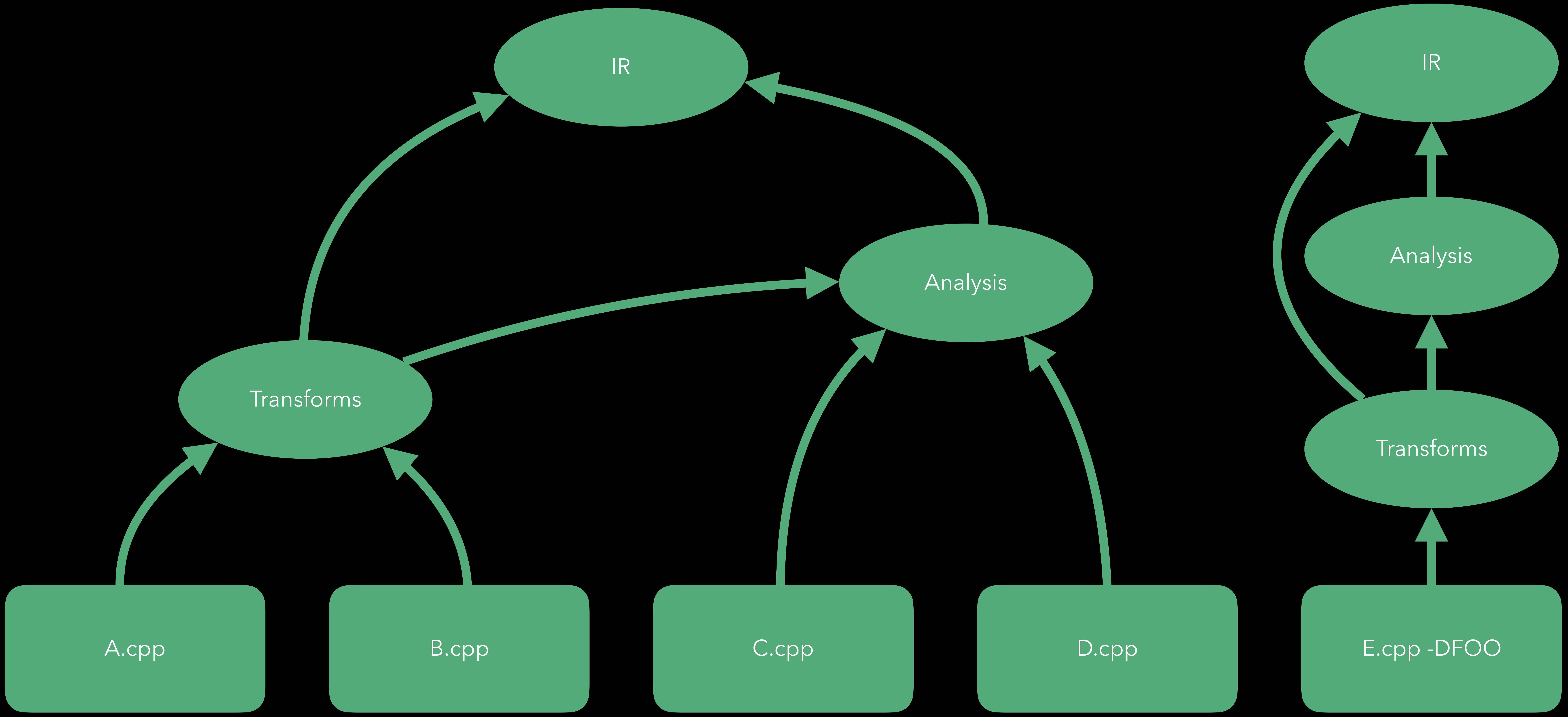
Implicit Modules

🔵 Compiler Discovered 🟢 Build System Known



Explicit Modules

🔵 Compiler Discovered 🟢 Build System Known



Explicit Clang Modules

- Better model: knowing modular dependencies before compiling
  - Allow more robust and reproducible builds 👍
  - Faster builds 🏎️
- Constraint: users shouldn’t have to specify modular dependencies
- Problem: which modules are needed?
- Solution: dependency discovery build phase for a build target

Clang Modules

Dependency Scanning

Fast Dependency Scanning

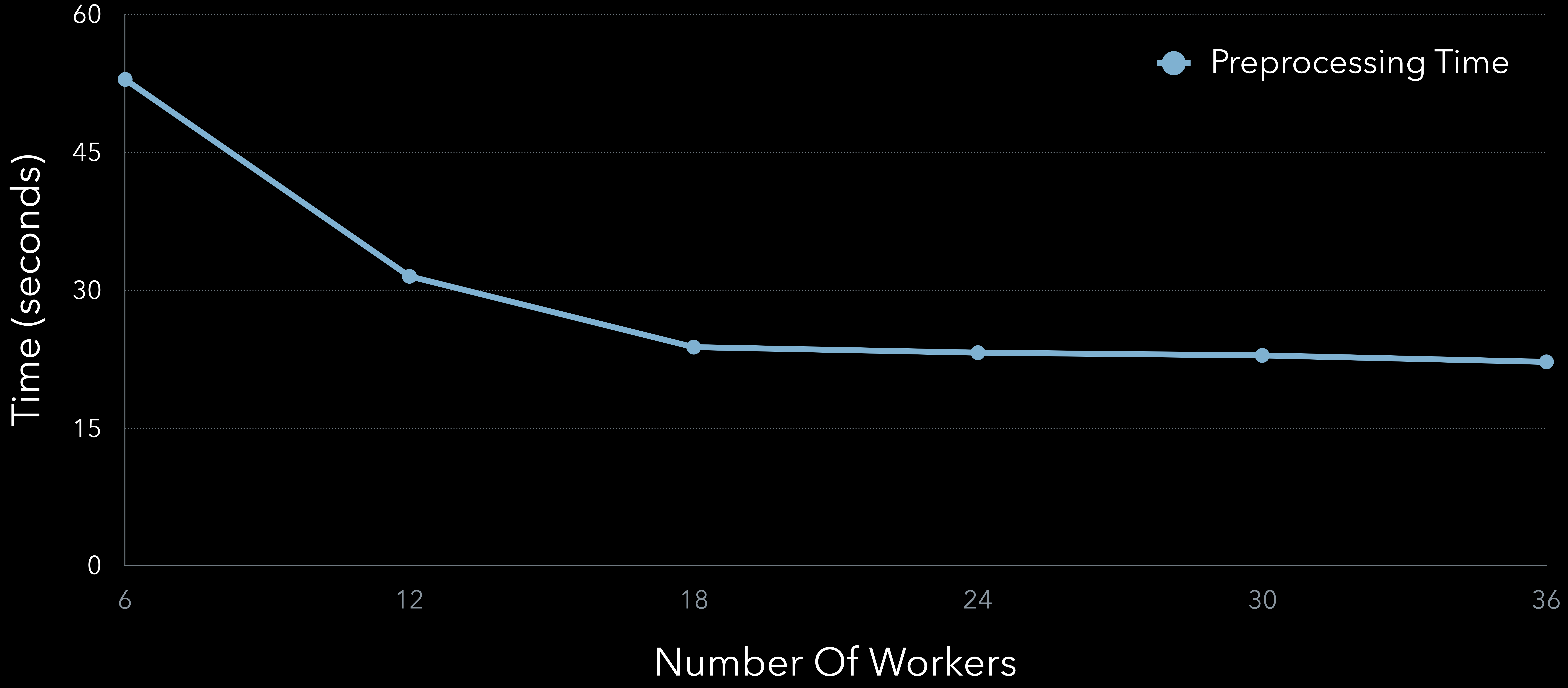
Dependency Extraction

Future Work

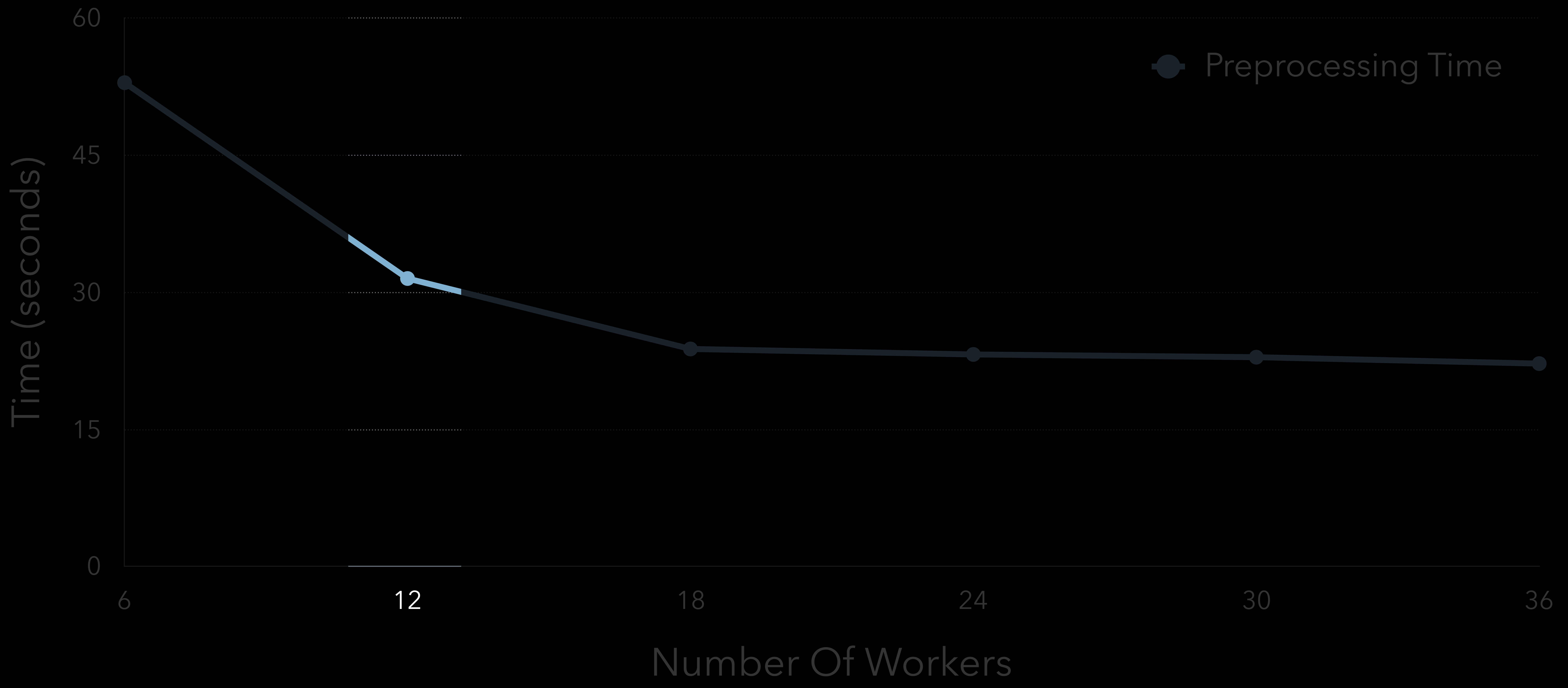
Canonical Dependency Scanning Phase

- Preprocess all translation units of a build target
- Write out included files into a .d
  - `clang -cc1 -Eonly -MT -dependency-file foo.d foo.c`
- How fast is the preprocessor?

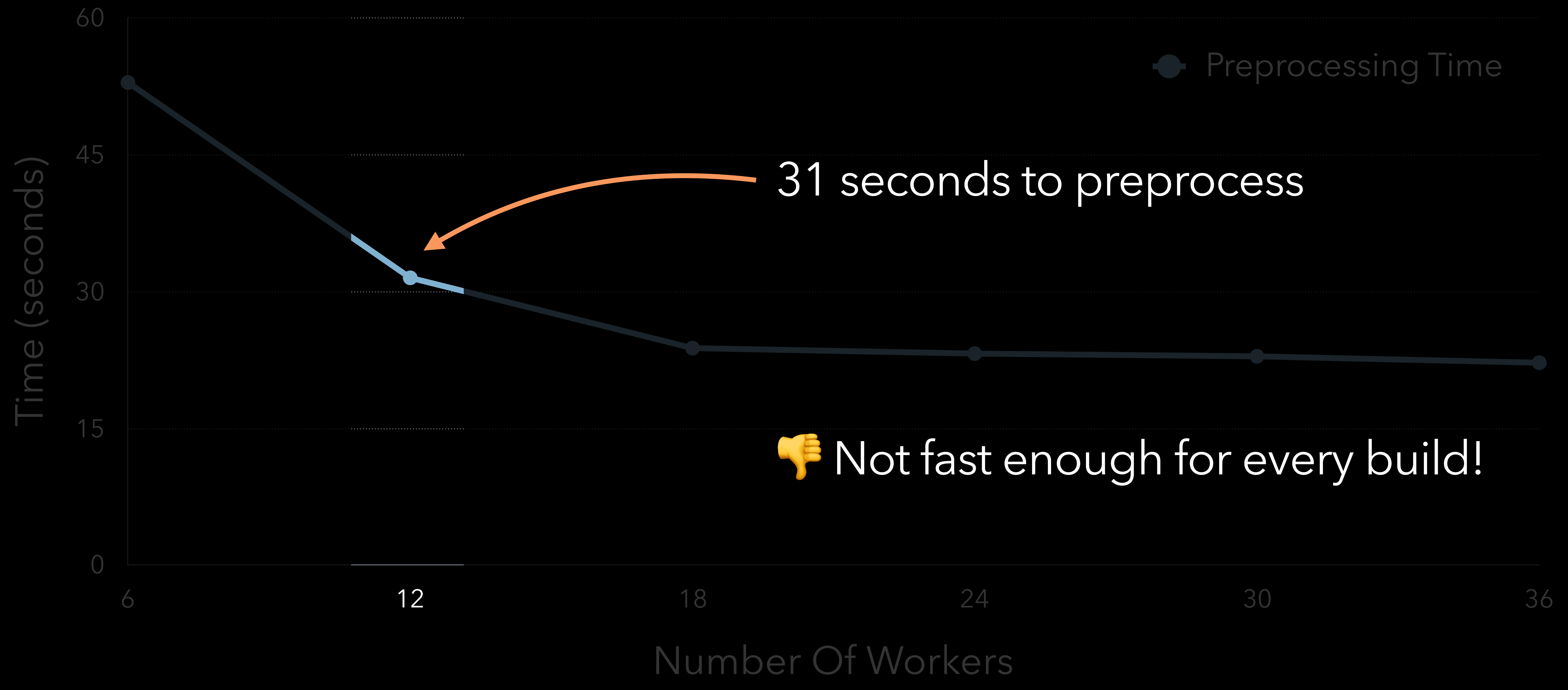
Clang and LLVM sources: preprocessing time on an 18-Core iMac Pro



Clang and LLVM sources: the 12 workers scenario



Clang and LLVM sources: the 12 workers scenario



31 seconds to preprocess

👎 Not fast enough for every build!

Clang Modules

Dependency Scanning

Fast Dependency Scanning

Dependency Extraction

Future Work

What Does The Preprocessor Do?

```
#ifndef HEADER_FILE
#define HEADER_FILE

#include “Compiler.h”

// Clang is an awesome tool!
class Clang: public Compiler {
public:
  void buildAllCode();
#ifndef NDEBUG
  void dump();
#endif
};

#endif
```

Lex tokens…

Evaluate `#ifndef` & `#define`

Lex more tokens…

Include “`Compiler.h`”

Lex more tokens…

Lex even more tokens 😫

Reducing Preprocessor Workload

```
#ifndef HEADER_FILE
#define HEADER_FILE

#include “Compiler.h”

// Clang is an awesome tool!
class Clang: public Compiler {
public:
  void buildAllCode();
#ifndef NDEBUG
  void dump();
#endif
};

#endif
```

Dependencies aren’t affected by these tokens

Reducing Preprocessor Workload

```
#ifndef HEADER_FILE
#define HEADER_FILE

#include “Compiler.h”

// Clang is an awesome tool!
class Clang: public Compiler {
public:
  void buildAllCode();
#ifndef NDEBUG
  void dump();
#endif
};

#endif
```

Dependencies aren’t affected by these tokens

Source Minimization

```
#ifndef HEADER_FILE
#define HEADER_FILE
#include “Compiler.h”
#endif
```

Source Minimization

```
#ifndef HEADER_FILE
#define HEADER_FILE
#include “Compiler.h”
#endif
```

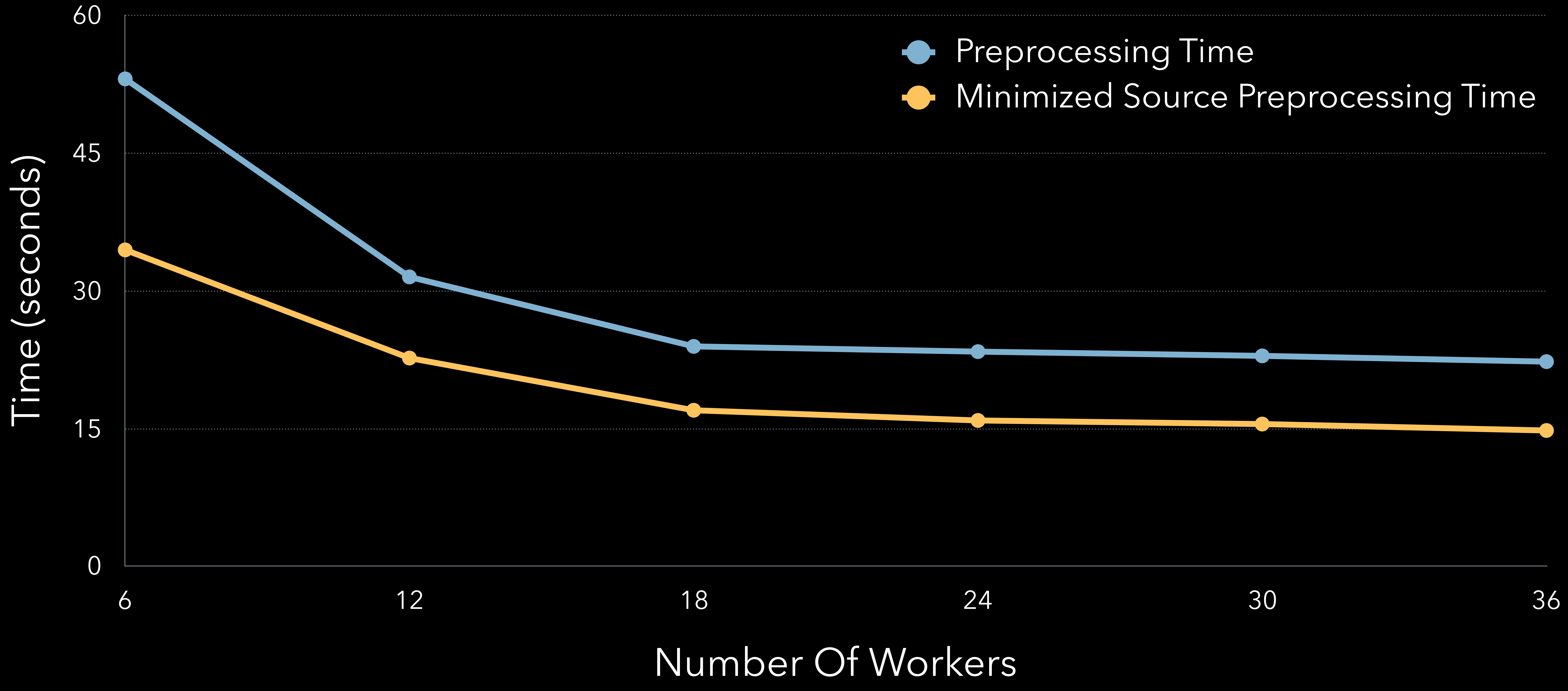
💡

Keep directives that may affect dependency list

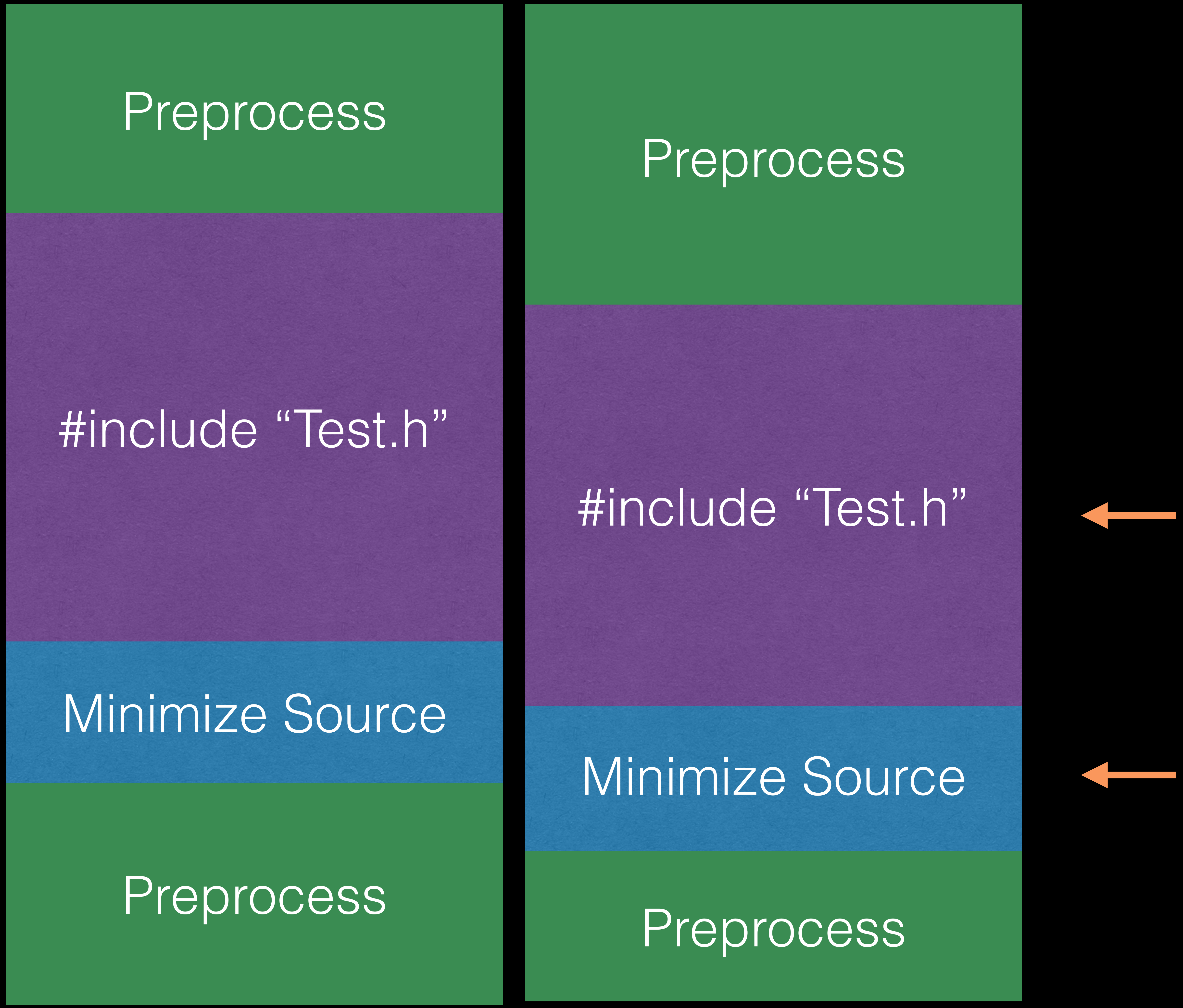
Strip everything else

Context free: source reused in any compilation

Clang and LLVM sources: 30% faster preprocessing



Problem: Clang Invocations



Parallel invocations do redundant work

Read the same file twice

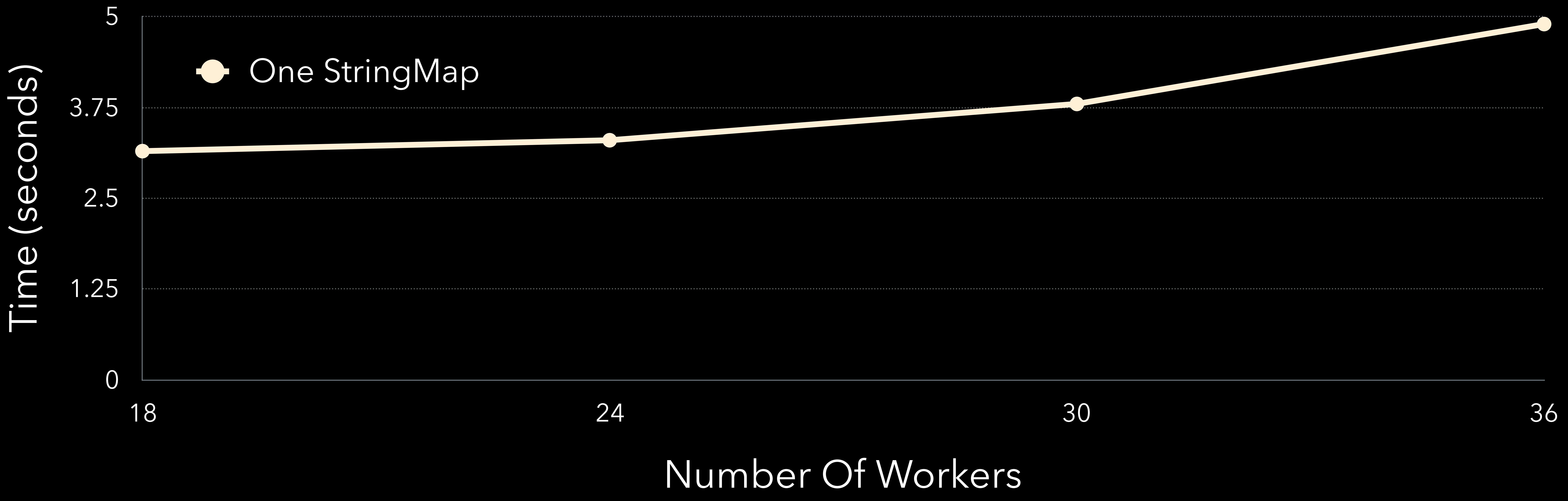
Minimize the same file twice

Introducing clang-scan-deps

- Library and command line tool for dependency scanning
  - Tool currently accepts compilation database and emits dependencies
- Runs preprocessor invocations in parallel
- Efficient: Reads and minimizes a source file only once
  - one shared `FileSystem` with shared minimized file cache
  - one shared `FileManager`

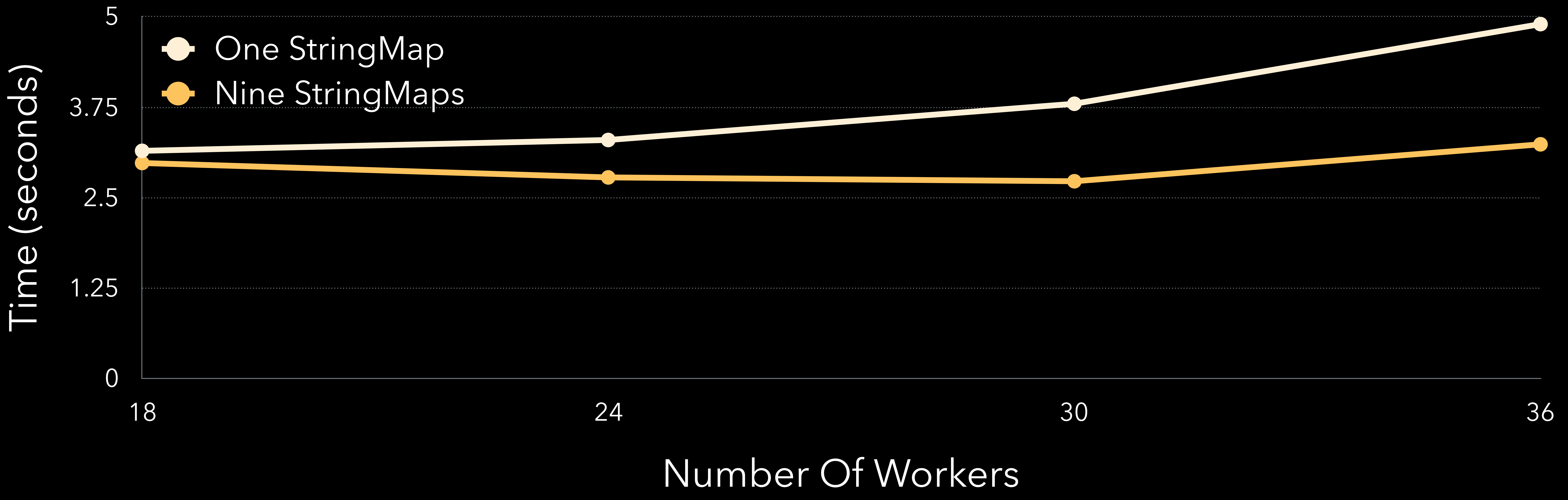
Minimized File Cache

- Maps from file name to cache entry
- Shared by worker threads: lock required access the `StringMap`
- High lock contention for many threads



Optimizing Minimized File Cache

- Solution: Array of `StringMap` addressed by hash of file name



Preprocessor Block Skipping

```
#ifdef NOT_TAKEN

// Important comment
#include “LexMeNot.h”

#elif

#include “IAmLexed.h”

#endif
```

When this **`#ifdef`** is not taken…

The tokens inside it are lexed…

Until the **`#elif`** is found

Took up to 10-15% of time in our profiles 😫

Optimizing Preprocessor Block Skipping

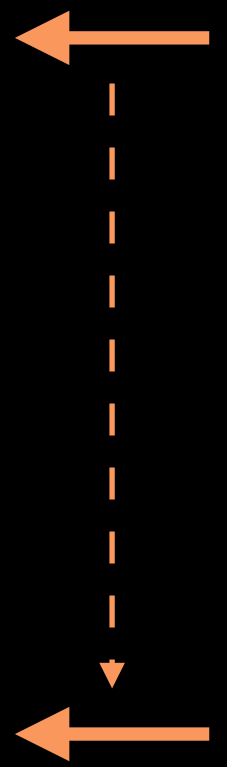
```
#ifdef NOT_TAKEN

// Important comment
#include “LexMeNot.h”

#elif

#include “IAmLexed.h”

#endif
```

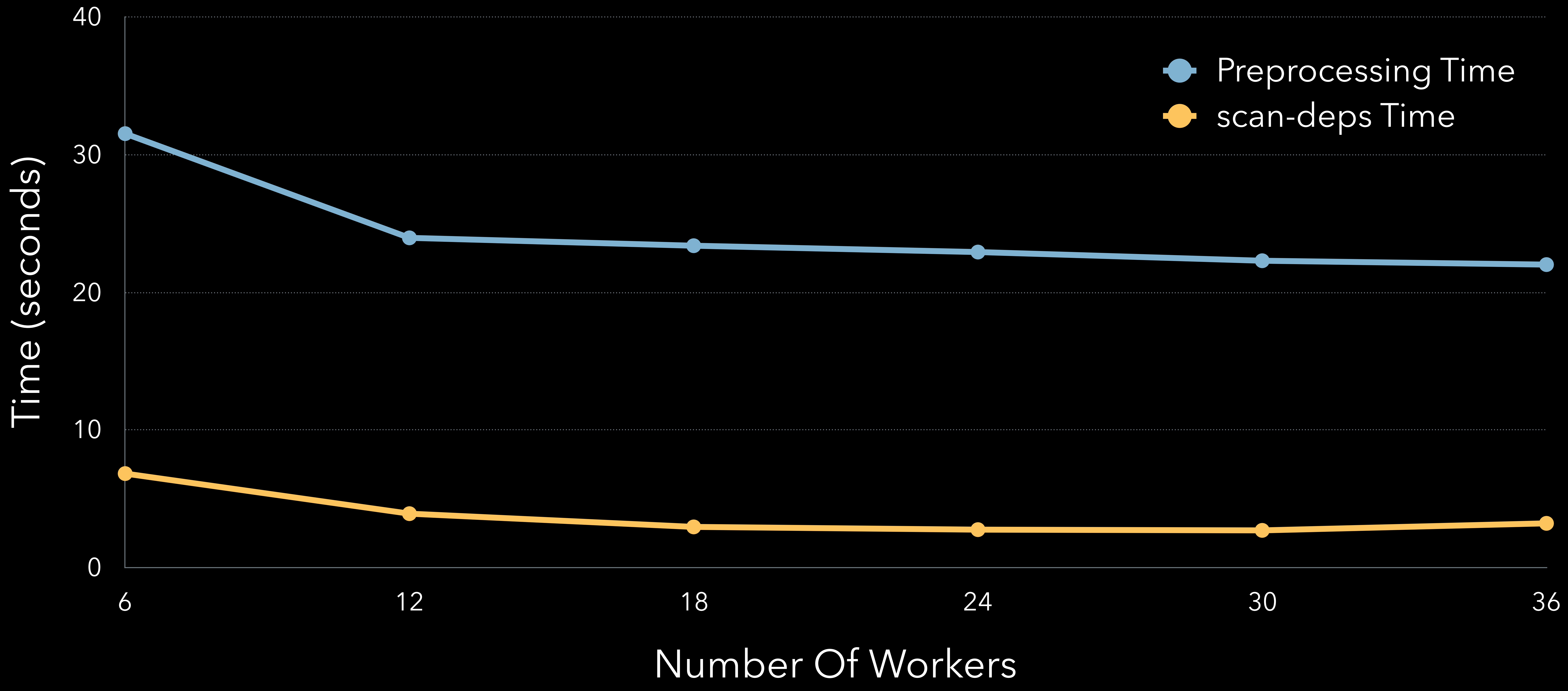


When this **`#ifdef`** is not taken…

Skip to **`#elif`**: add offset to Lexer’s pointer

Offset computed when minimizing file 😃

Clang and LLVM sources: 5-10x faster dependency scanning



Things We Aren’t Going To Support

```
#define AT_IMPORT @import

AT_IMPORT Foundation;

#define WHY(X) _##X ("clang module import X")

WHY(Pragma);
```

➡ We want to disallow this behavior in Clang

Modular Dependencies

- clang-scan-deps builds implicit modules with minimized files
  - For now still uses old implicit module build machinery 🙈
- Dependencies are extracted from the fast implicit build

Clang Modules

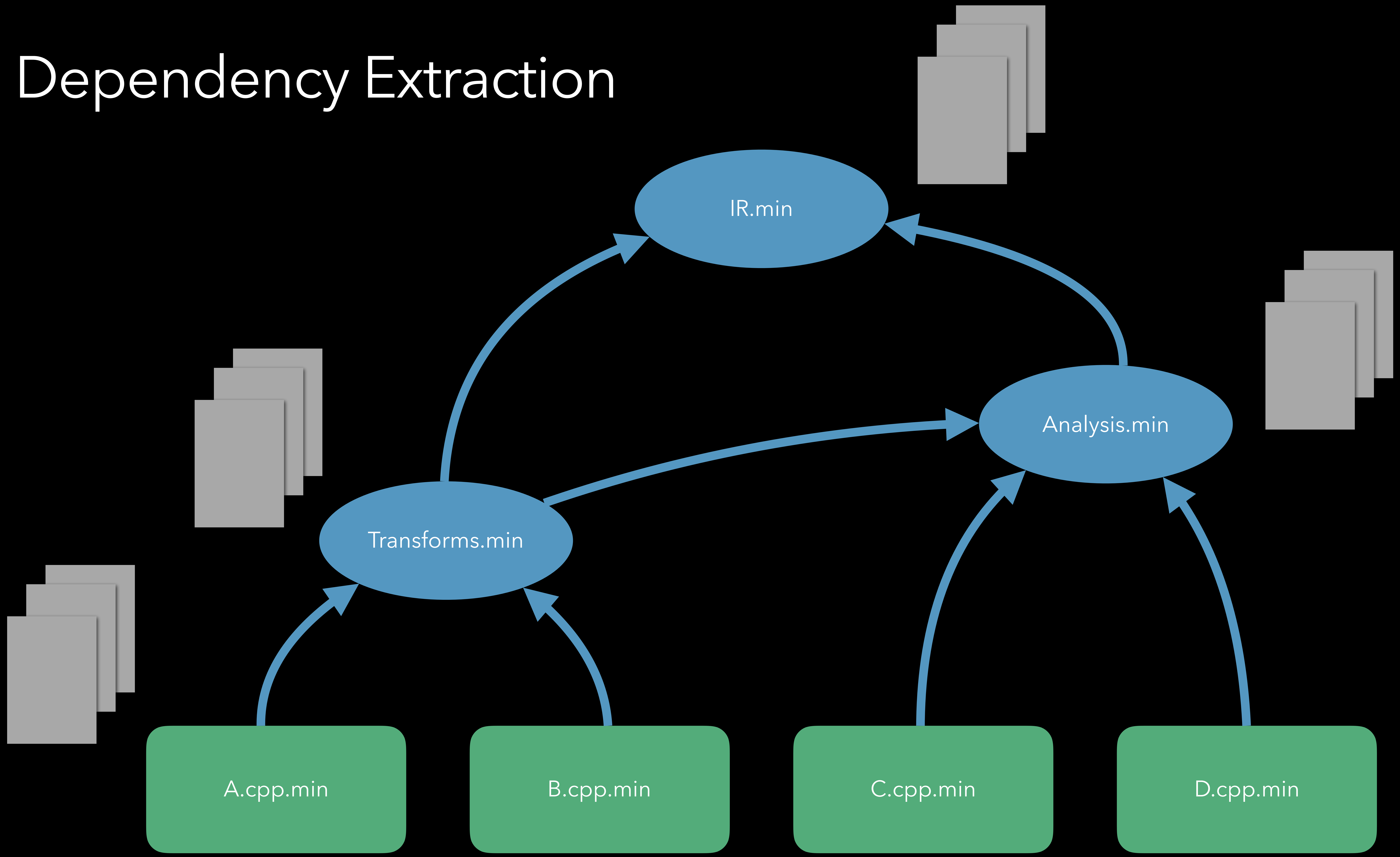
Dependency Scanning

Fast Dependency Scanning

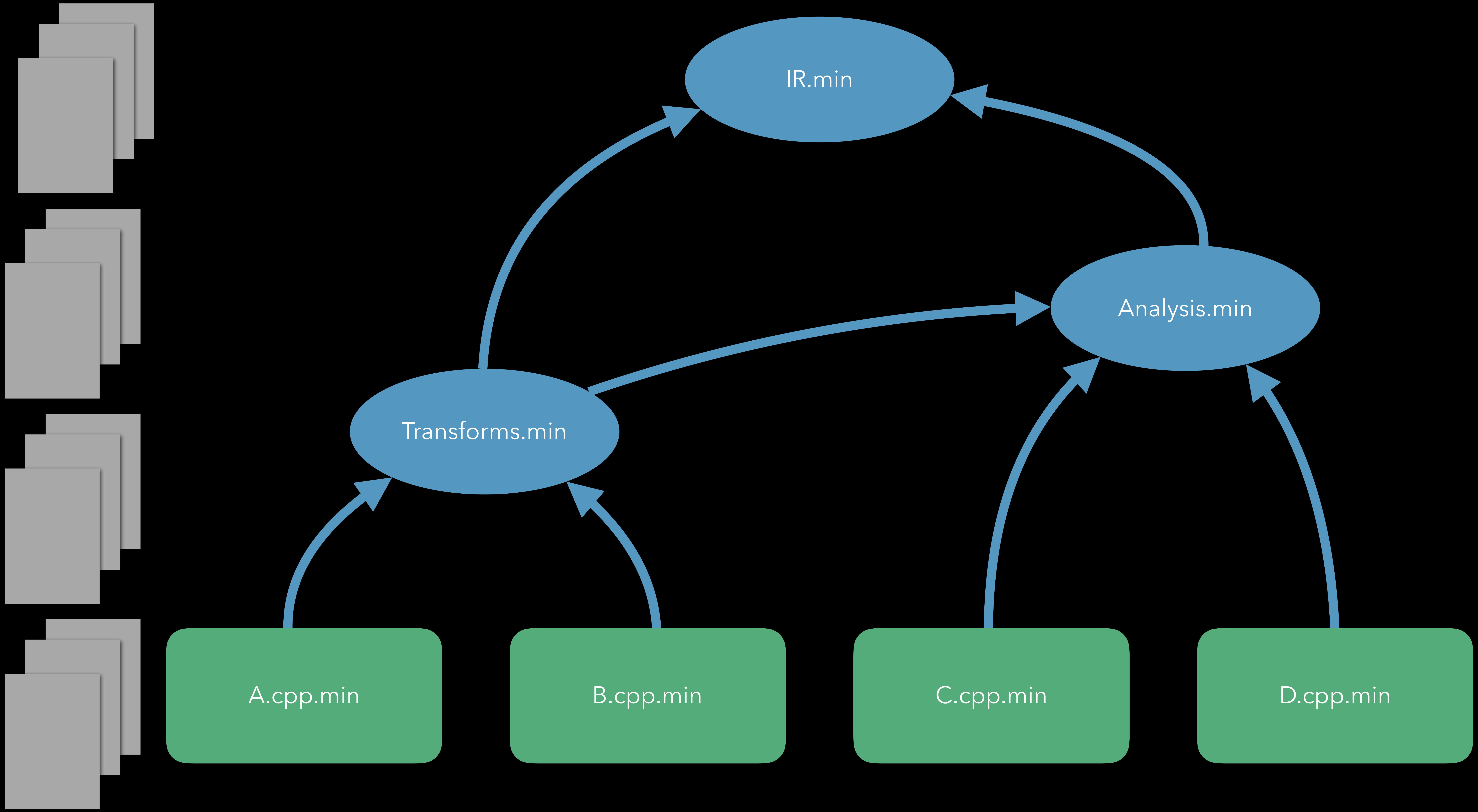
Dependency Extraction

Future Work

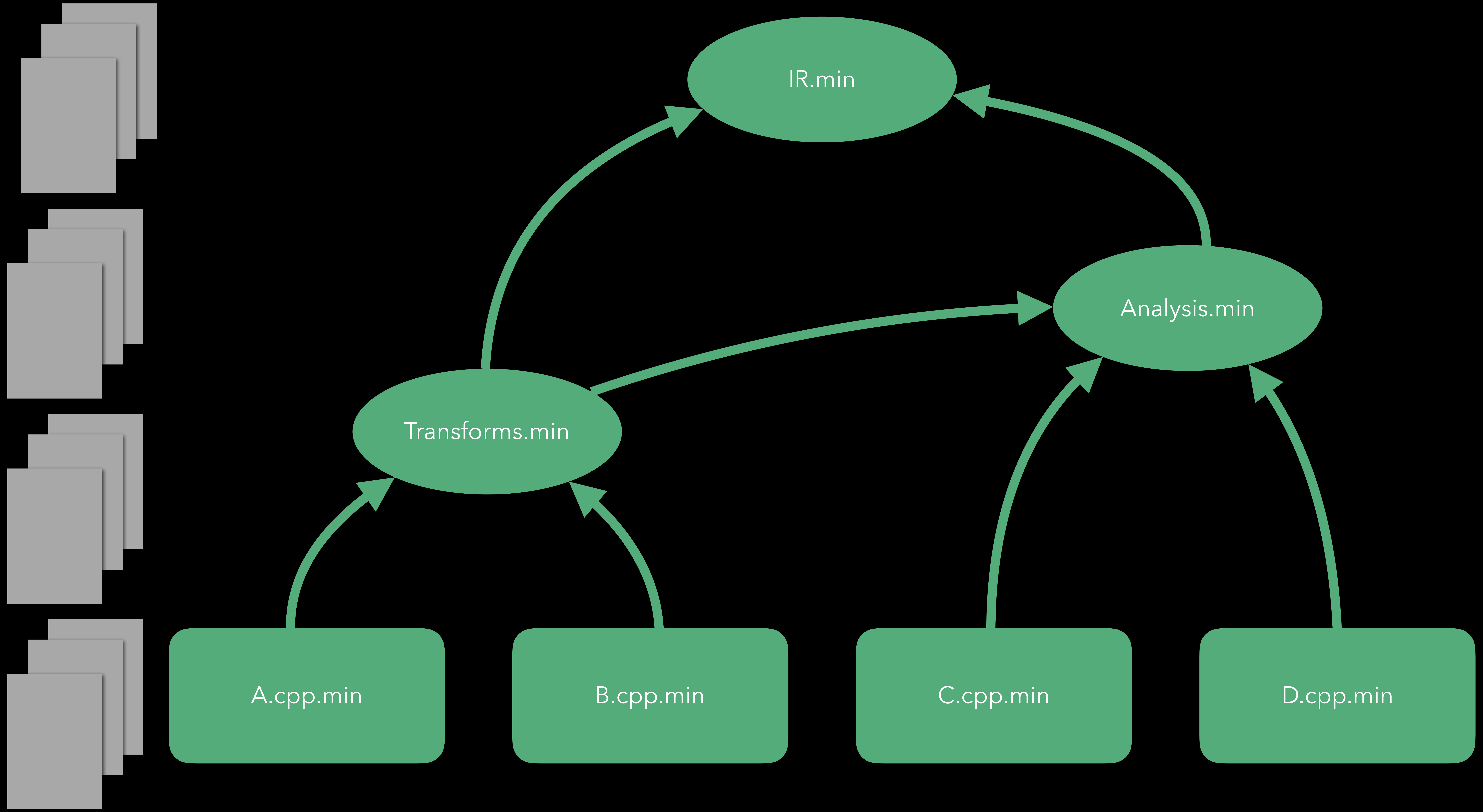
Dependency Extraction



Dependency Extraction



Dependency Extraction

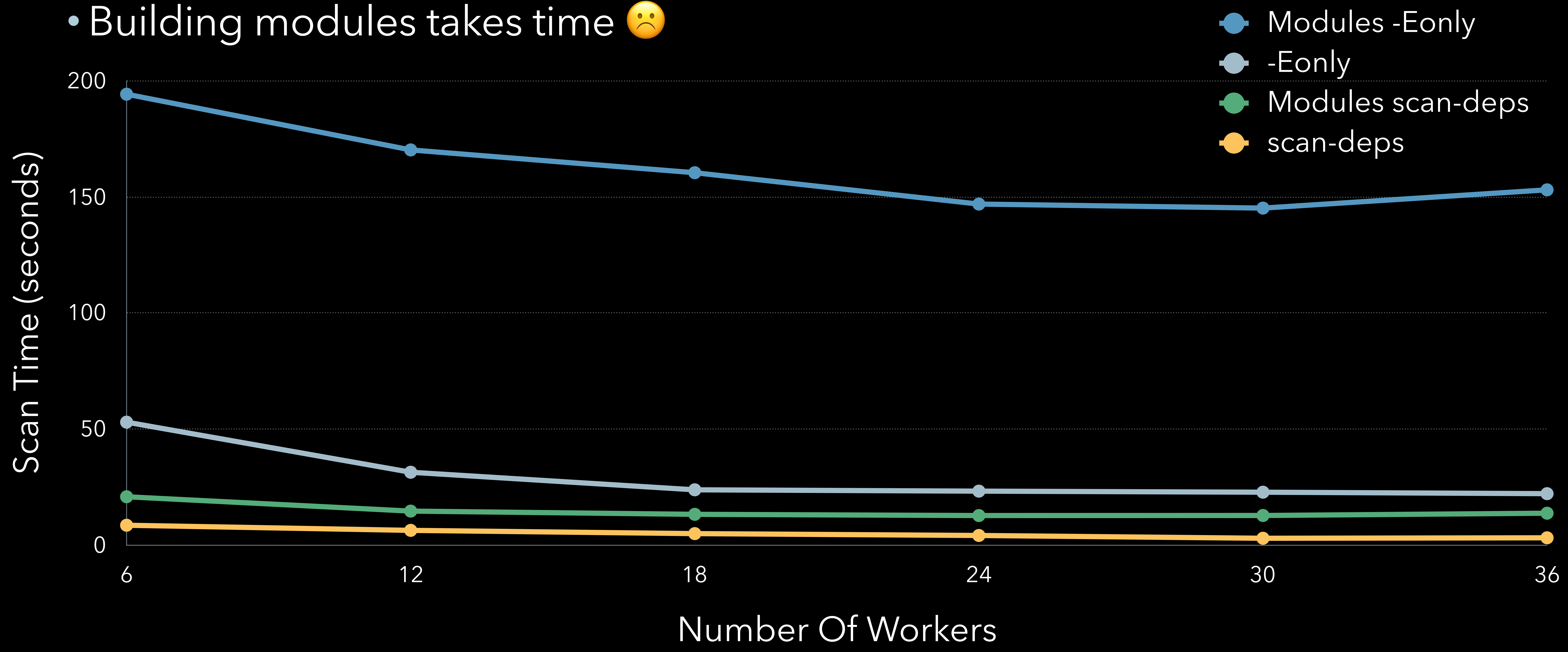


Dependency Extraction

```
build LLVM_Transforms.pcm: cxx_explicit_module
    llvm/include/llvm/module.modulemap | LLVM_IR.pcm LLVM_Analysis.pcm std.pcm
  module_id = LLVM_Transforms
  moduledeps = -fmodule-file=LLVM_Config_Config.pcm -fmodule-file=std.pcm
               -fmodule-file=LLVM_IR.pcm -fmodule-file=LLVM_Analysis.pcm
  args = builds/release/bin/clang++ -cc1 -fmodules ...
```

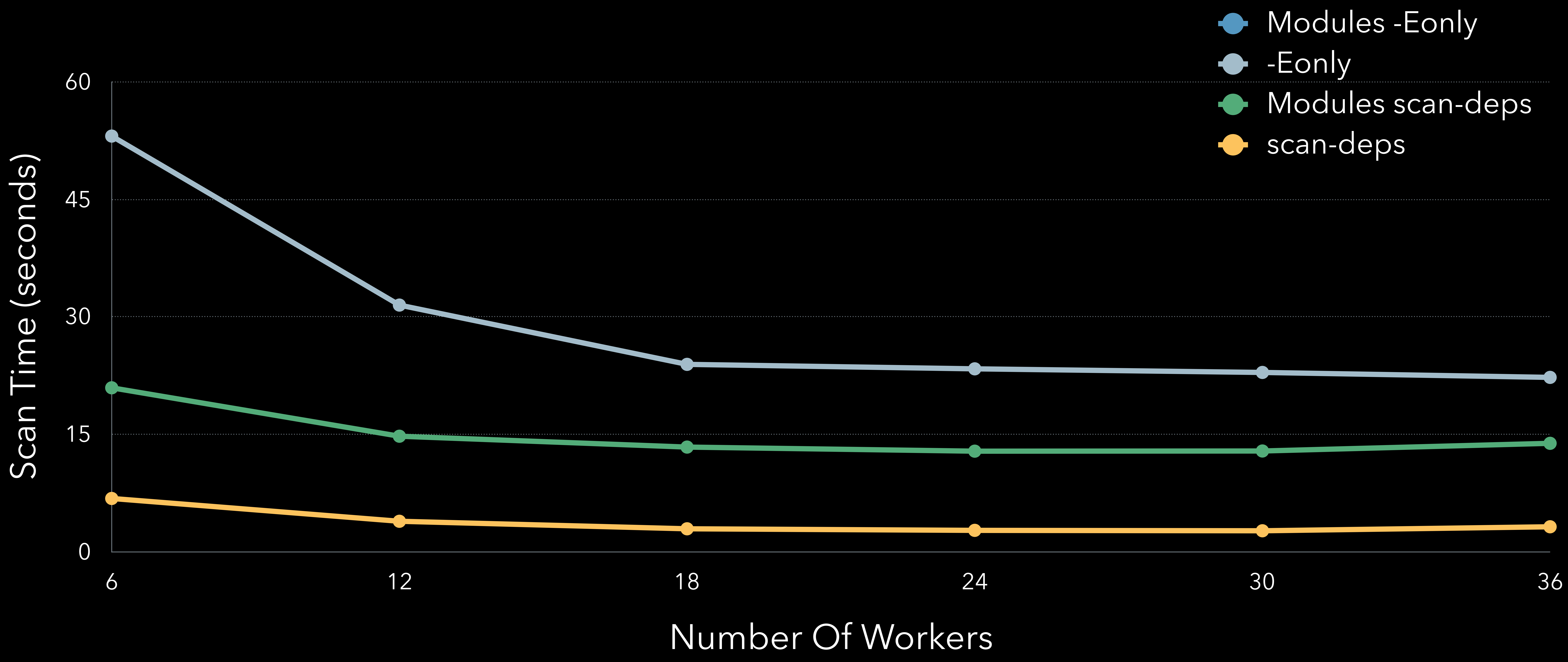
Initial Results - Scanning

- Faster than modules -Eonly and -Eonly, but slower than scan-deps
- Building modules takes time ☹️

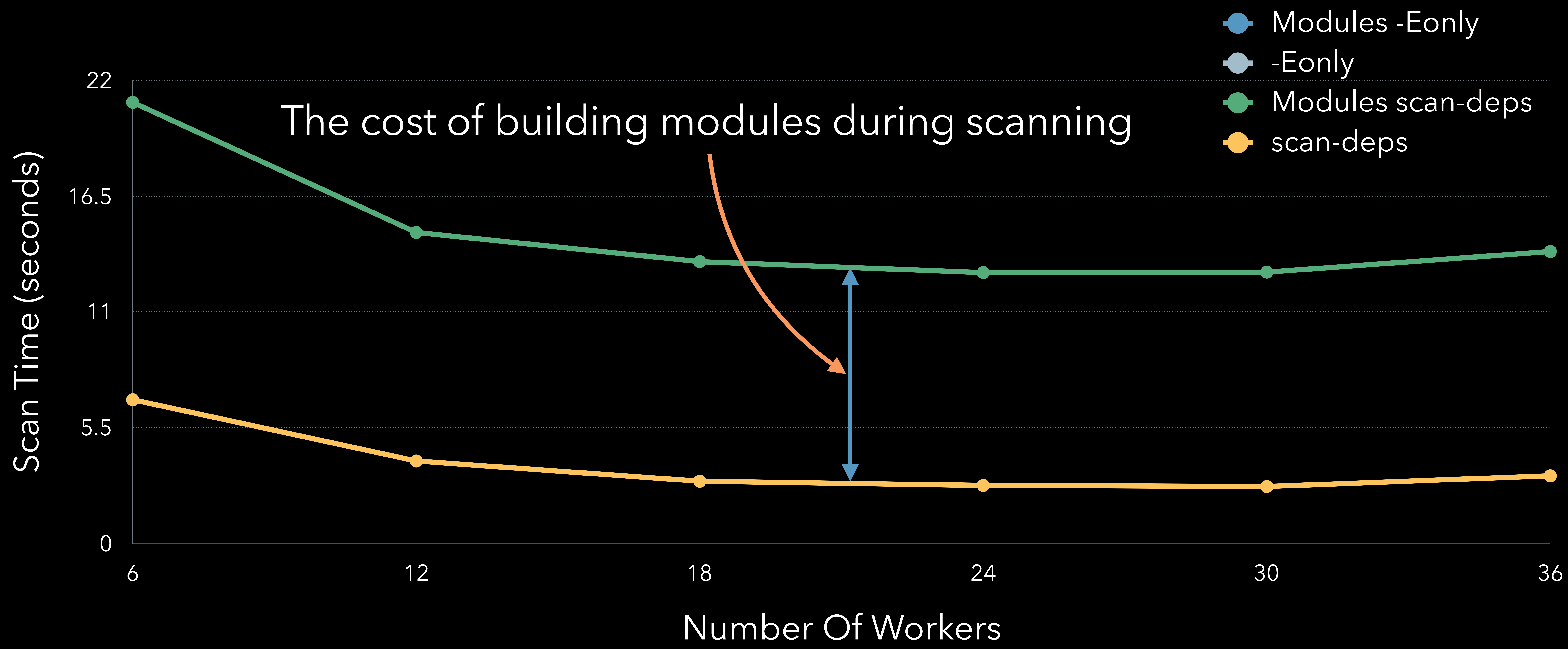


Initial Results - Scanning

- Faster than modules -Eonly and -Eonly, but slower than scan-deps



Initial Results - Scanning



The cost of building modules during scanning

Initial Results - Building

- About 5-15% speedup on an 18-Core iMac Pro

Implicit
Scan + Explicit
Explicit
1300
975
650
325
0
Build Time (seconds)
6
12
18
24
30
36
Number Of Workers

Bugs 🐞

- Implicit and Explicit modules behave differently
- Different ideas about textual headers vs. modular headers
    - Changes dependencies
- Implicit creation of module maps for frameworks
- Different code paths

Clang Modules

Dependency Scanning

Fast Dependency Scanning

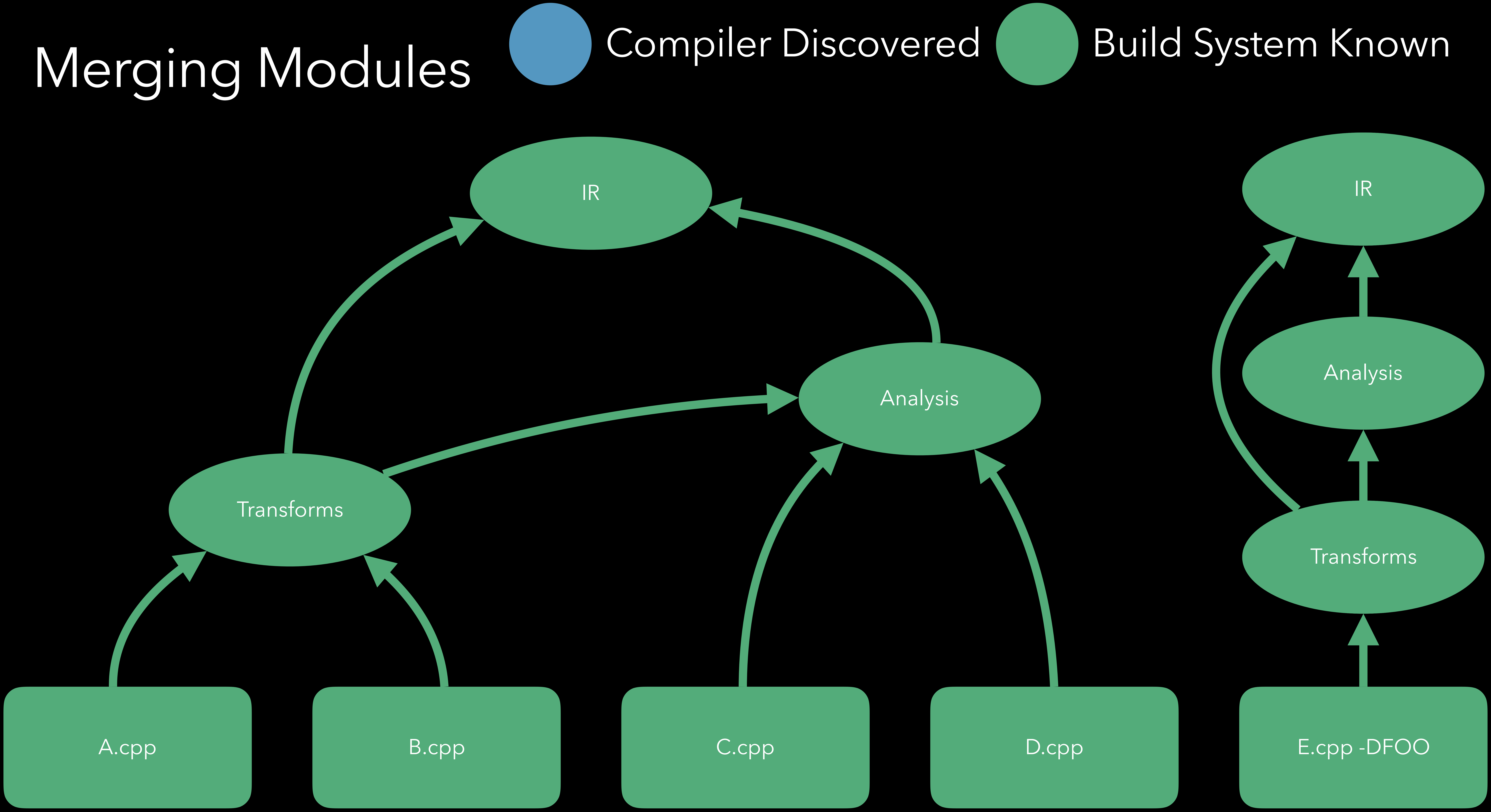
Dependency Extraction

Future Work

Future Work

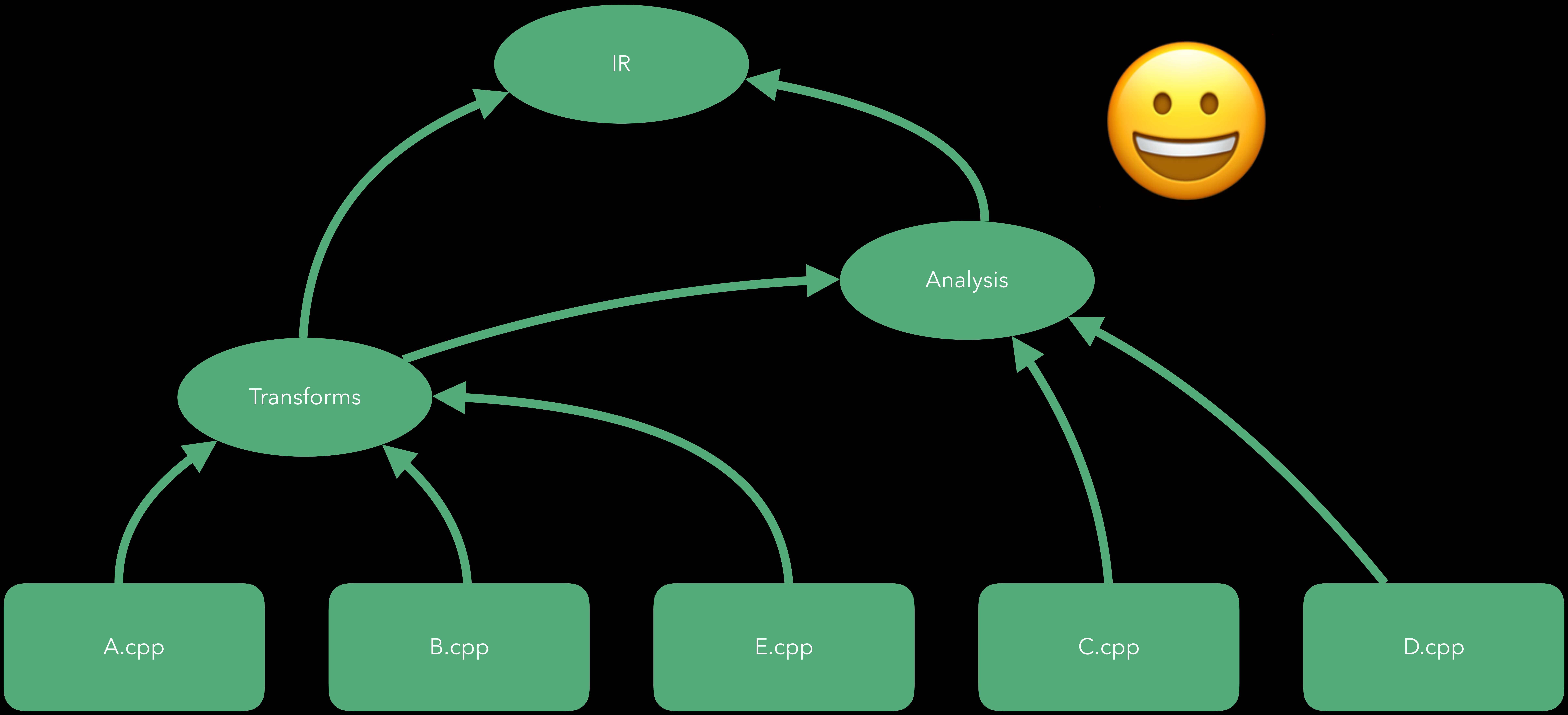
- Optimize
  - Don’t build modules, just need deps
  - Cache results, don’t write to disk
  - Incremental
  - Merge the build graph for compatible modules

Merging Modules



Merging Modules

● Compiler Discovered ● Build System Known



Future Optimization

Future Work

- Optimize
  - Don’t build modules, just need deps
  - Cache results, don’t write to disk
  - Incremental
  - Merge the build graph for compatible modules
- C++20 Modules
  - Support for `import module` and `import <header>`
- Upstream
  - Patches: https://reviews.llvm.org/D55463, D60233

Questions?

clang-scan-deps – Fast Dependency Scanning For Explicit Modules
Alex Lorenz, Michael Spencer, LLVM Developers' Meeting, Brussels, Belgium, April 2019