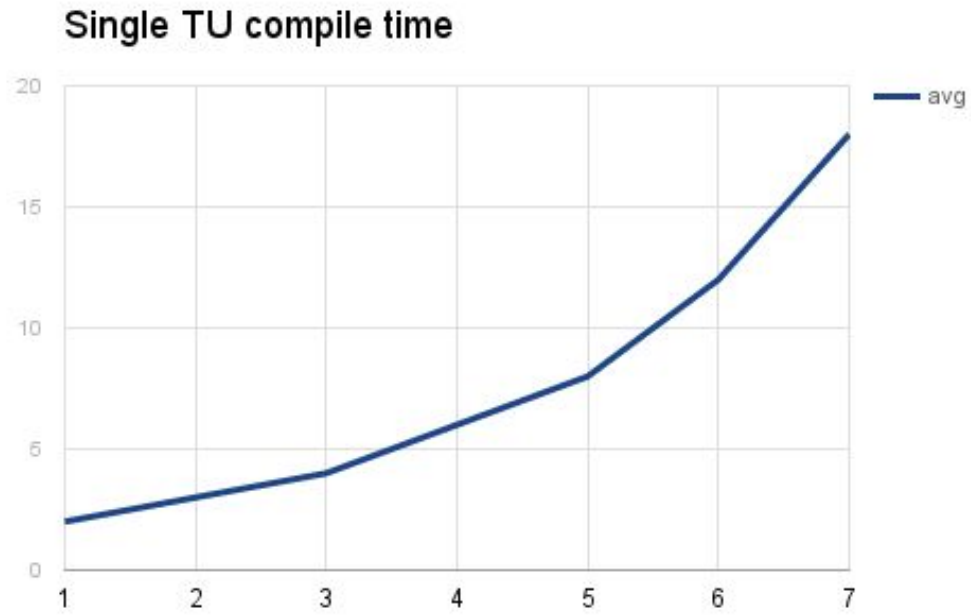
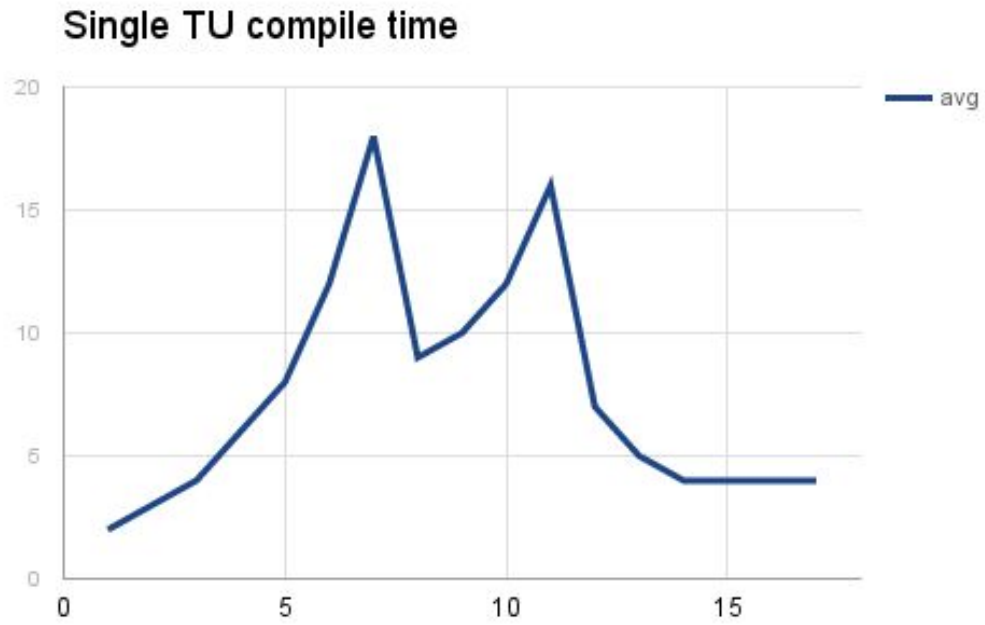
Deploying C++ modules to 100s of millions of lines of code

Manuel Klimek (klimek@google.com)
CppCon 2016


Single TU compile time
20
15
10
5
0
1
2
3
4
5
6
7
avg


Single TU compile time
20
15
10
5
0
0
5
10
15
avg

The lay of the land

The Google Codebase

- multiple 100 MLOC C++ code
- ~same amount of generated code
- continuously integrated
- distributed and caching build system

Protocol Buffers

foo.proto:

```
message Foo {
  optional int32 bar = 1;
}
```

foo.pb.h:

```
class Foo {
  int32 bar();
  void set_bar(int32);
  ...
}
```

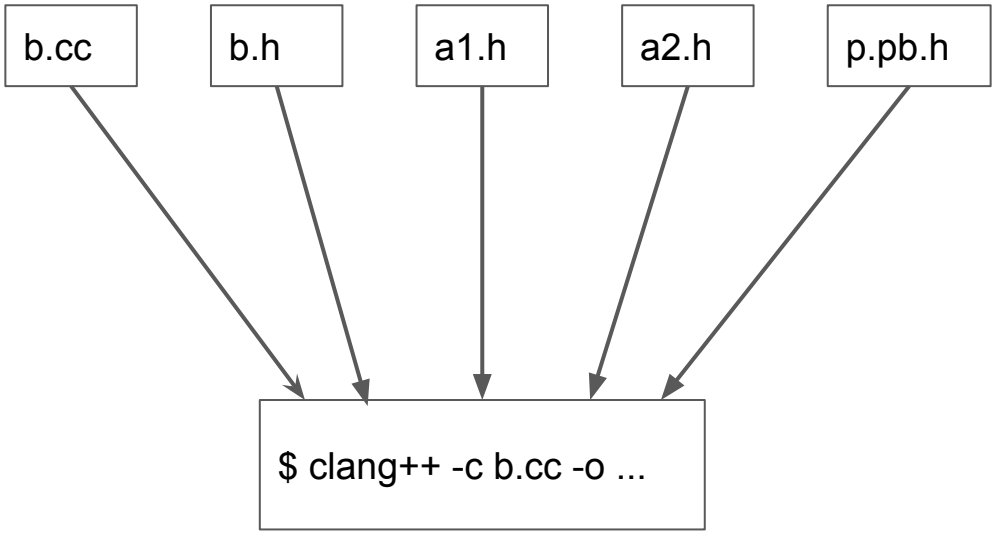
The build system

```
proto_library(name="p", srcs=["p.proto"])

cc_library(name="a", hdrs=["a1.h", "a2.h"],
           srcs=["a.cc"])

cc_library(name="b", hdrs=["b.h"], srcs=["b.cc"],
           deps=["a", "p"])
```

b.cc:

```
#include "a1.h"
#include "b.h"
#include "other.h"
#include "p.pb.h"
```

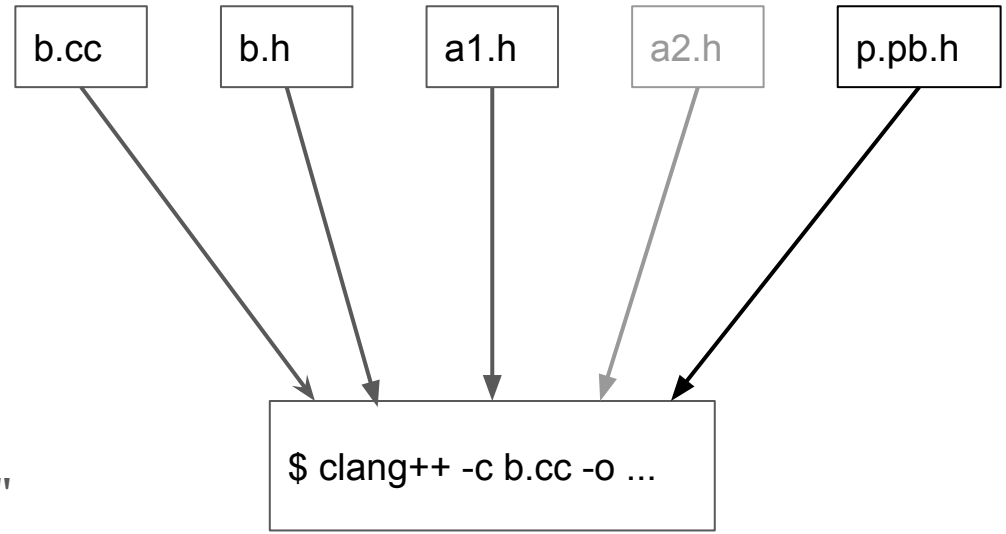
Send each file to distributed system to compile



b.cc:

```
#include "a1.h"
#include "b.h"
#include "other.h"
#include "p.pb.h"
```

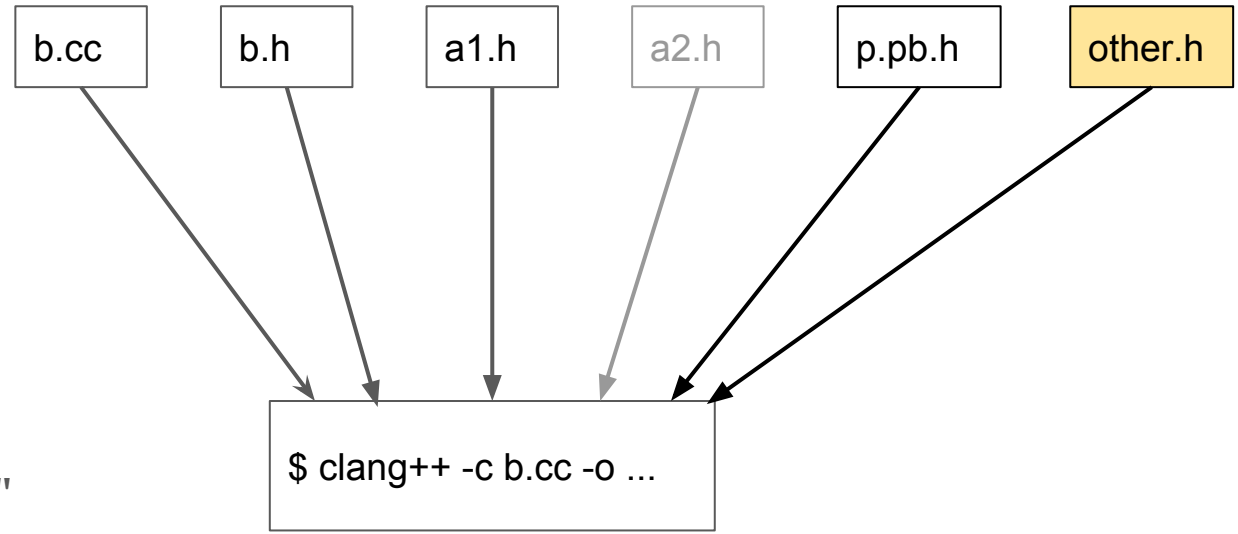
Scan code and prune unnecessary headers



b.cc:

```
#include "a1.h"
#include "b.h"
#include "other.h"
#include "p.pb.h"
```

Scan code and add missing headers



b.cc:

```
#include "a1.h"
#include "b.h"
#include "other.h"
#include "p.pb.h"
```

Modules? What modules?

- Clang's pre-TS modules implementation

- **no** changes to **syntax**

- changes to the **semantics**

Why not wait for modules and change all code?

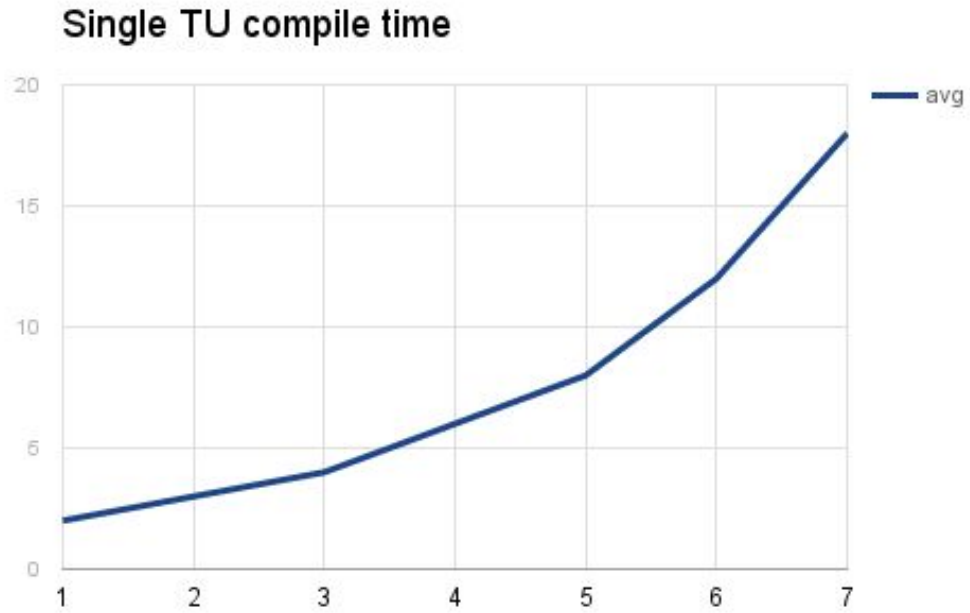
- multiple 100 MLOC

- gain implementation insights

- back when we started, there was no modules TS proposal

- we had an acute problem

How bad did it get?



https://xkcd.com/303/

What happened?

- was optimization getting slower? no

- was clang or gcc getting slower? no

- #lines of transitive include closure growing super-linearly!
(and the compilation scales mostly linearly with LOC)

- some TUs with > 10 MLOC transitive `#include` closure

Protocol buffers

a.proto:

```
message A {
  optional string name = 1;
}
```

a.pb.h

```
class A {
  bool has_name() const;
  void clear_name();
  static const int kNameFieldNumber = 1;
  const string& name() const;
  void set_name(const string& v);
  void set_name(const char* v);
  void set_name(const char* v, size_t s);
  string* mutable_name();
  string* release_name();
  void set_allocated_name(string* name);
};

inline bool A::has_name() const { ... }
inline void A::clear_name() { ... }
inline const string& A::name() const { ... }
inline void A::set_name(const string& v) { ... }
inline void A::set_name(const char* v) { ... }
inline void A::set_name(const char* v, size_t s) { ... }
inline string* A::mutable_name() { ... }
inline string* A::release_name() { ... }
inline void A::set_allocated_name(string* name) { ... }
```

Protocol buffers (solved!)

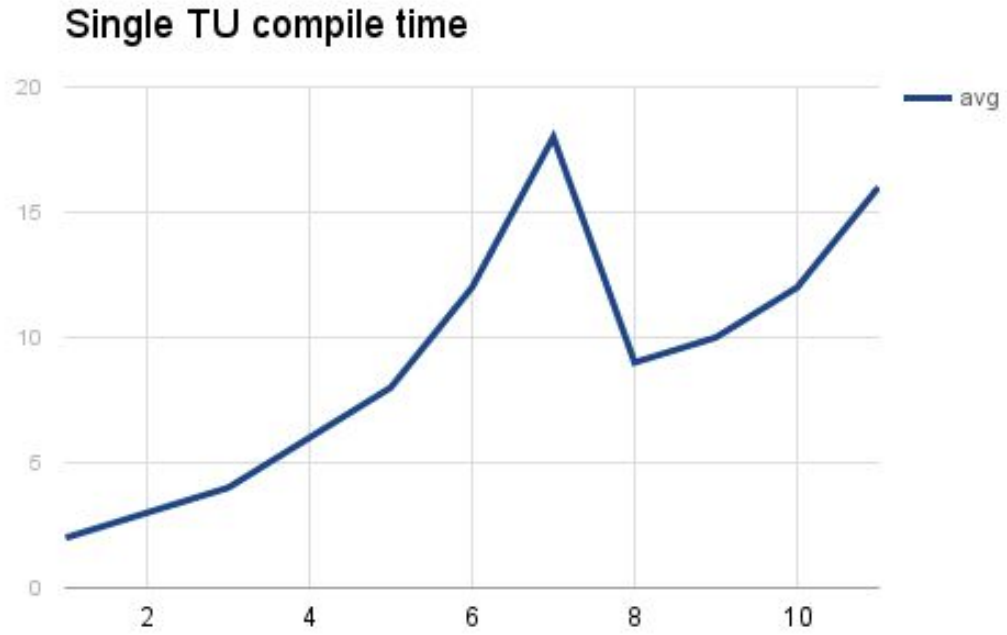
a.proto:

```
message A {
  optional string name = 1;
}
```

a.pb.h

```
class A {
  bool has_name() const;
  void clear_name();
  static const int kNameFieldNumber = 1;
  const string& name() const;
  void set_name(const string& v);
  void set_name(const char* v);
  void set_name(const char* v, size_t s);
  string* mutable_name();
  string* release_name();
  void set_allocated_name(string* name);
};
#ifdef NDEBUG
inline bool A::has_name() const { ... }
inline void A::clear_name() { ... }
inline const string& A::name() const { ... }
inline void A::set_name(const string& v) { ... }
inline void A::set_name(const char* v) { ... }
inline void A::set_name(const char* v, size_t s) { ... }
inline string* A::mutable_name() { ... }
inline string* A::release_name() { ... }
inline void A::set_allocated_name(string* name) { ... }
#endif /*NDEBUG*/
```

Solved! Solved?



Addressing the root cause (attempt 1)

- generated code needs to be **fast** (inline everything!)

b.pb.h:

```
#include "a.pb.h"

class B {
  ...
  A* mutable_a() {
    if (a_ == nullptr) a_ = new A();
    return a_;
  }
};
```

Addressing the root cause (attempt 2)

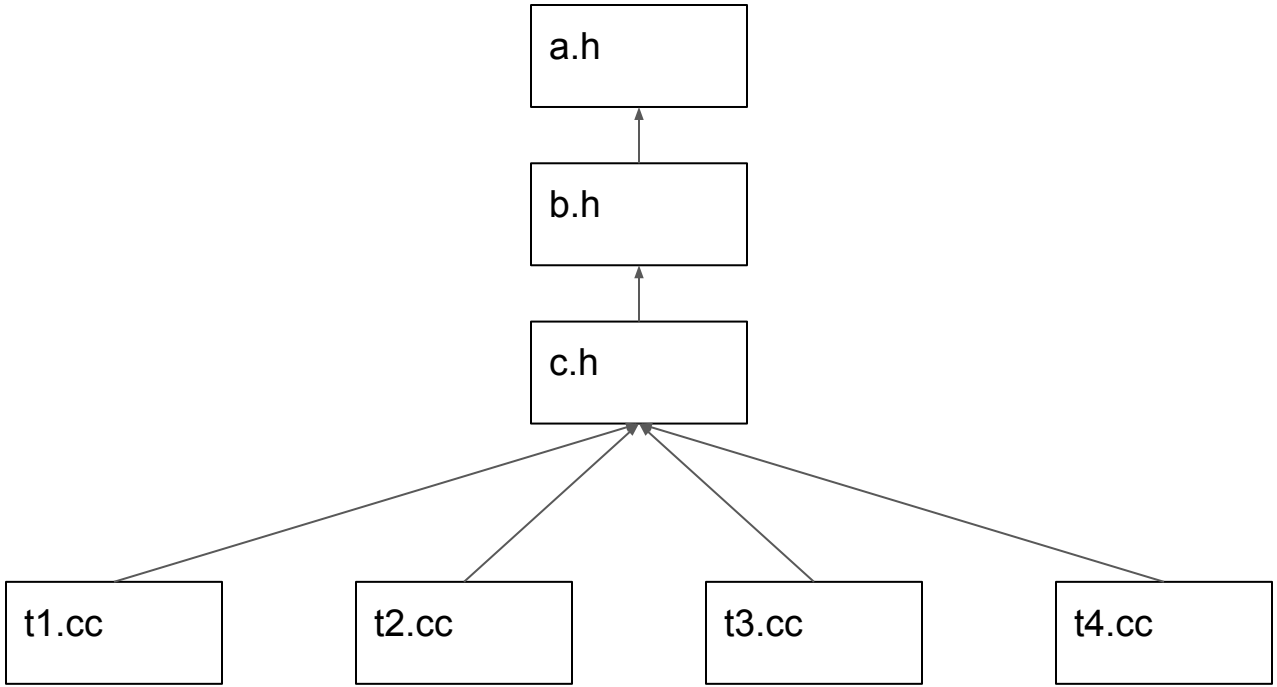
- need a compilation model that fundamentally doesn't scale superlinearly

- store AST and lazily load symbols

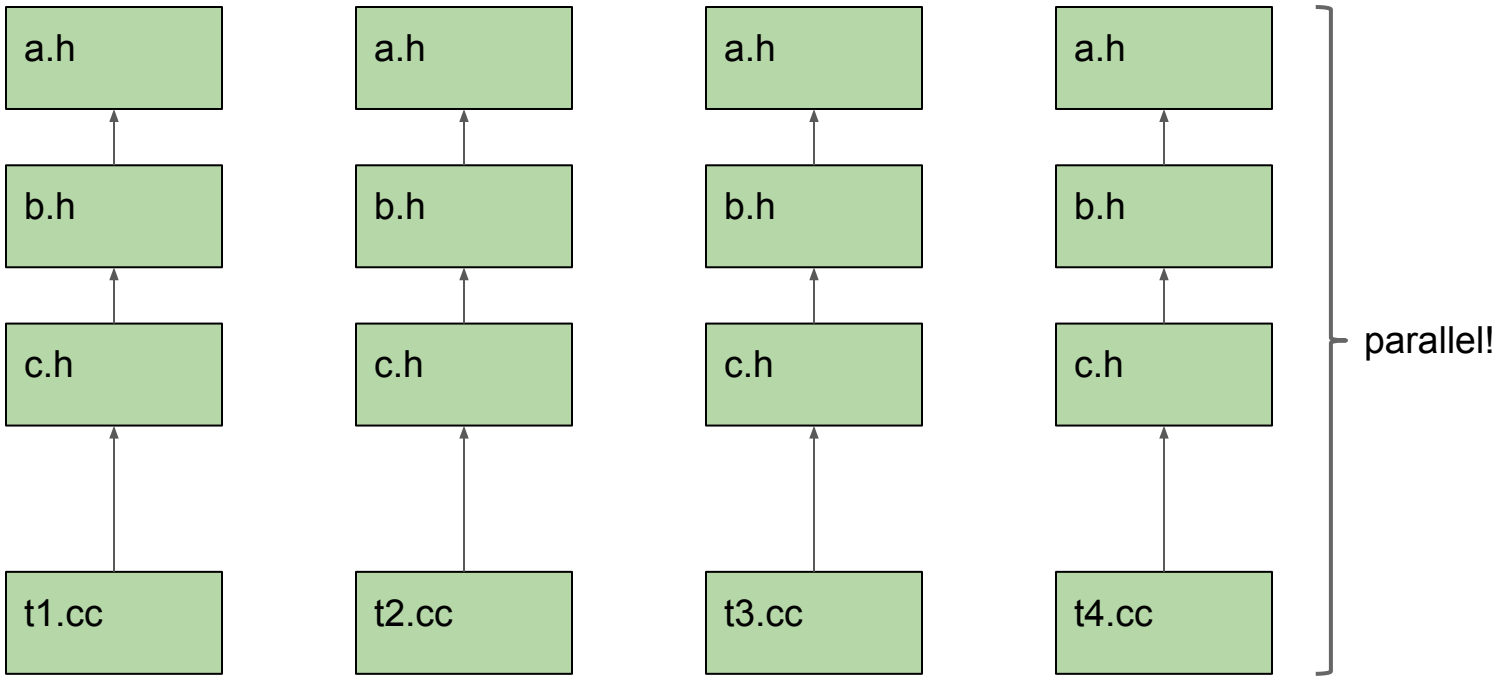
-> modules!

Let's predict what will happen...

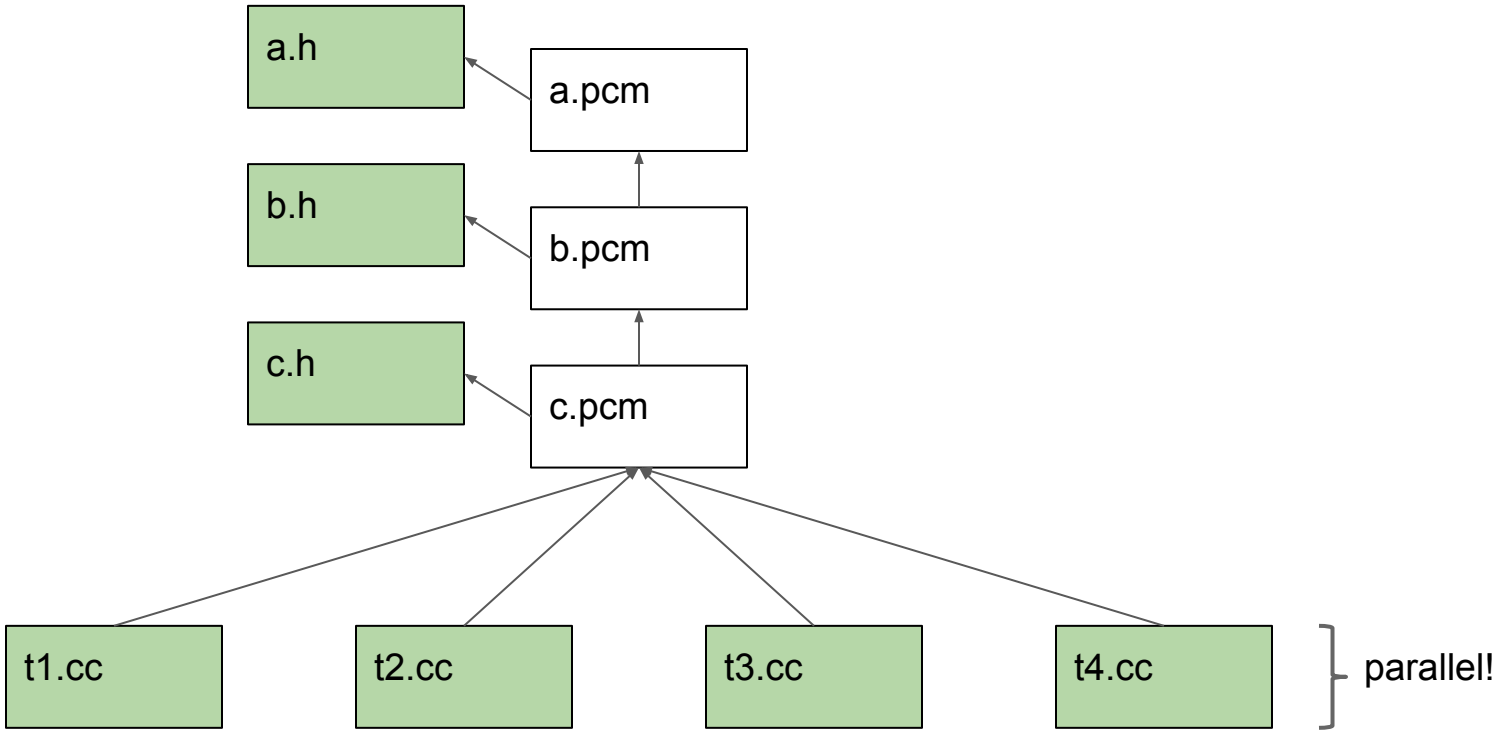
Exciting: a shiny new compilation model for C++



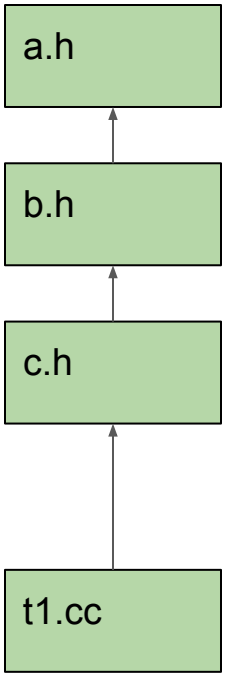
Exciting: a shiny new compilation model for C++



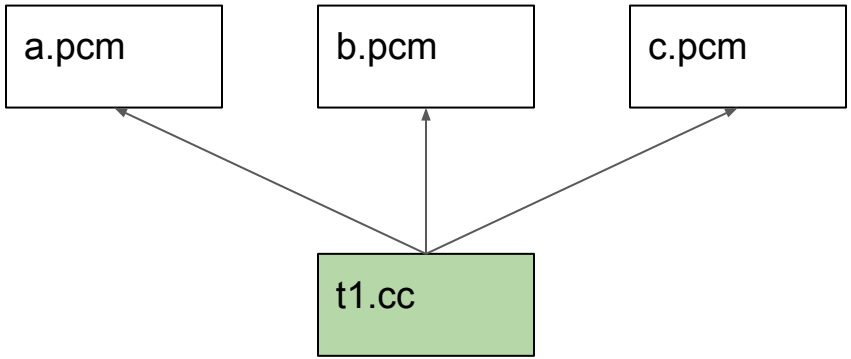
Exciting: a shiny new compilation model for C++



Exciting: a shiny new compilation model for C++



Exciting: a shiny new compilation model for C++



Exciting: a shiny new compilation model for C++

-> Longer critical path

-> Speedup of incremental compiles

-> Less CPU use overall


WHERE DO WE START

Selecting a subproblem

- protocol buffers:
    - generated code, controlled environment
    - largest problem

Describing a module to clang

```
proto_library(name="b", srcs=["b.proto"], deps=["a"])
```

```
module "b" {
  export *
  header "b.pb.h"
  use "a"
  use "stl"
}
```

Non-modular headers

```
module "b" {
  export *
  header "b.pb.h" -> #include "message.h"
  use "message"
  use "stl"
}

module "message" {
  textual header "message.h"
}
```

Semantic changes

- headers must compile on their own

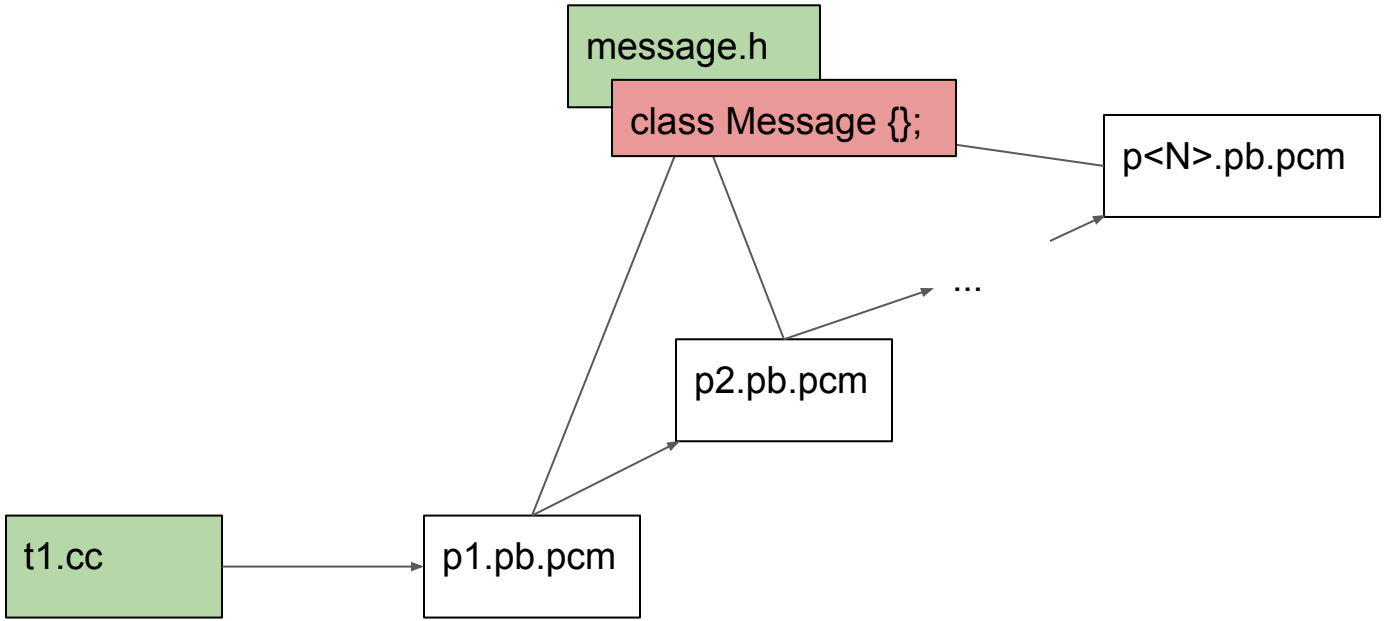
- more & different ODR violations diagnosed

-> headers must mean the same everywhere they're included from

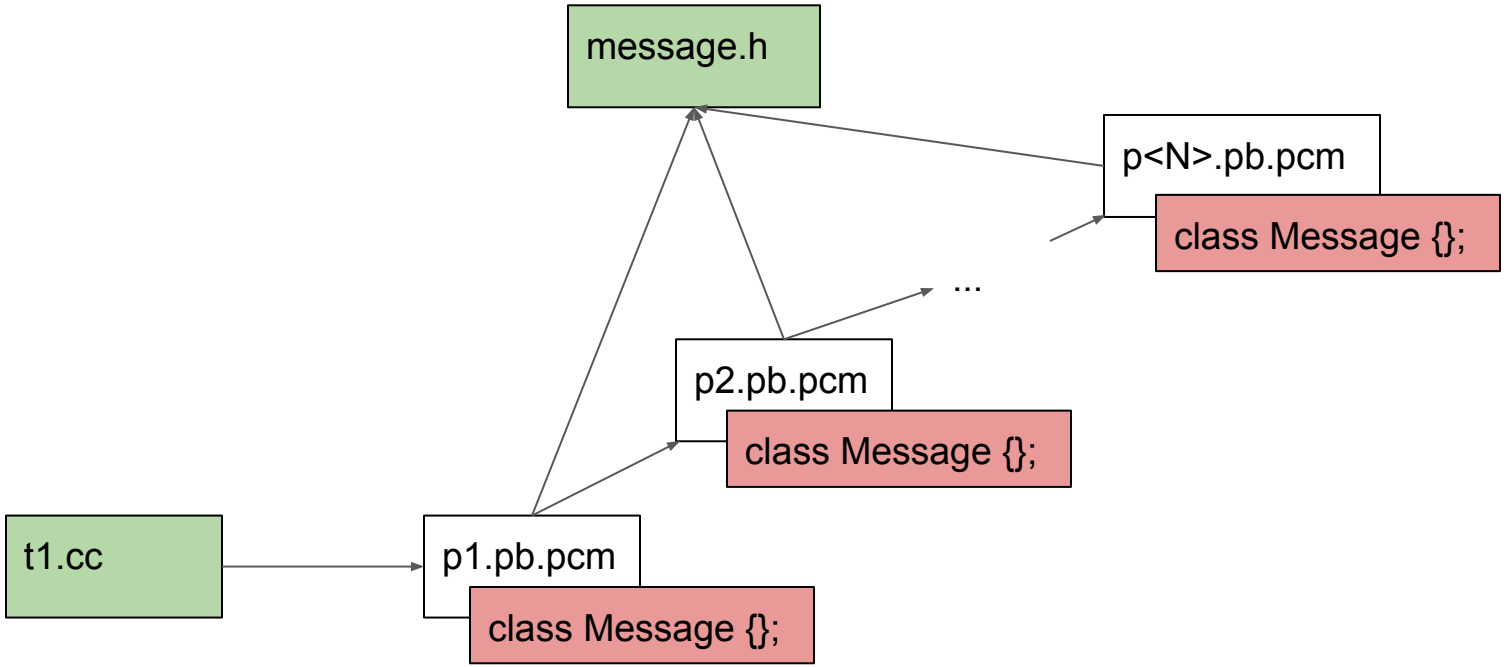

THINGS THAT BREAK

Things that break: modules twice as slow!

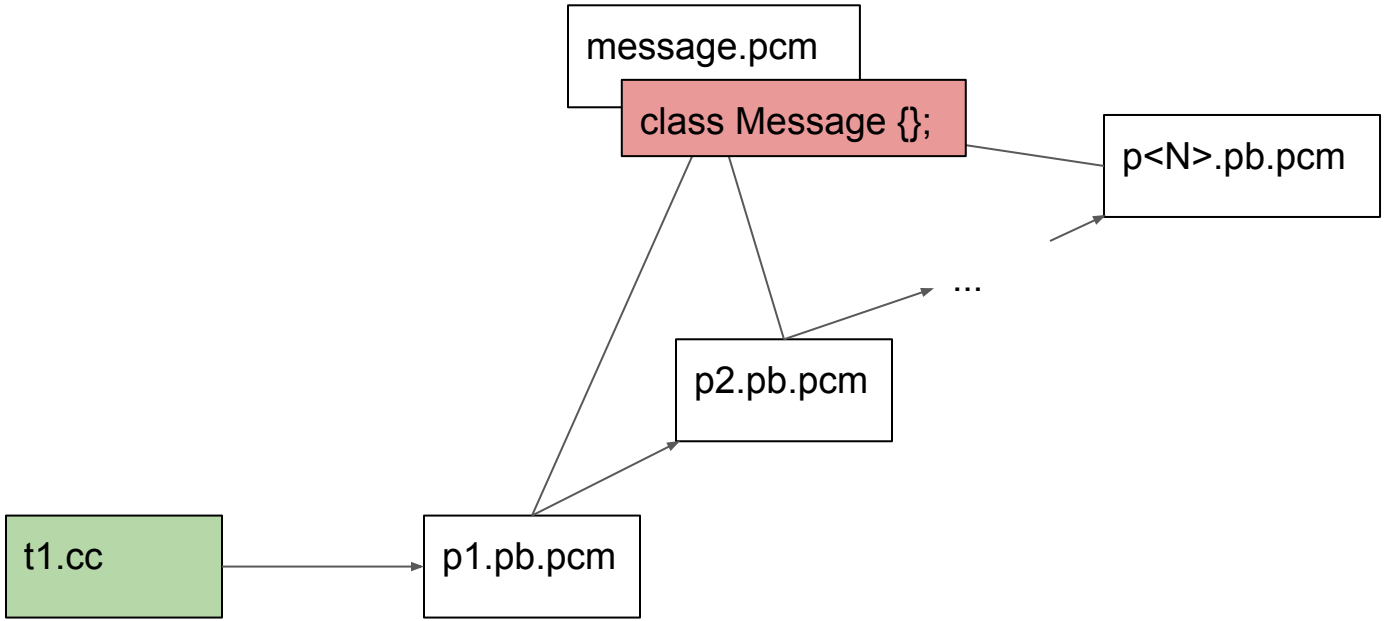
Things that break: modules twice as slow!



Things that break: modules twice as slow!



Things that break: modules twice as slow!



Problem: non-modular headers

-> performance problems when depending on many proto libraries (merging)

-> each module contains a copy of the standard library, ...

-> some need for bottom-up modularization

Things that break: relying on #include order...

a.h:

```
int a();
```

b.h:

```
inline int b() { return a(); }
```

c.cc:

```
#include "a.h"
#include "b.h"

int main() { return b(); }
```

Things that break: _impl.h files

a.h:

```
struct C { int a(); };
...
#include "a_impl.h"
```

a_impl.h:

```
inline int C::a() { return 42; }
```

Things that break: C header

a.cc:

```
extern "C" {
#include "b.h"
}
```

Things that break: __MODULE__

- `__MODULE__` used by Arm C++ compiler conflicts with Clang's definition

Things that break: command line size

- explicit module files on command line cause command lines above the limits

-> pass only "top-level" module files

-> the calculation of those files makes the build system slower

Things that break: mixing configuration macros

a.h:

```
#include <cassert>
inline int a(int i) {
  assert(i > 0);
  return i+2;
}
```

b.cc:

```
#define NDEBUG
#include "a.h"
int main() { return a(-42); }
```

Configuration management

- need different module for each configuration

- theoretically: exponential explosion; in reality, a couple of configurations

- switch off modules when requiring different configurations
    -> automatically ignore modules

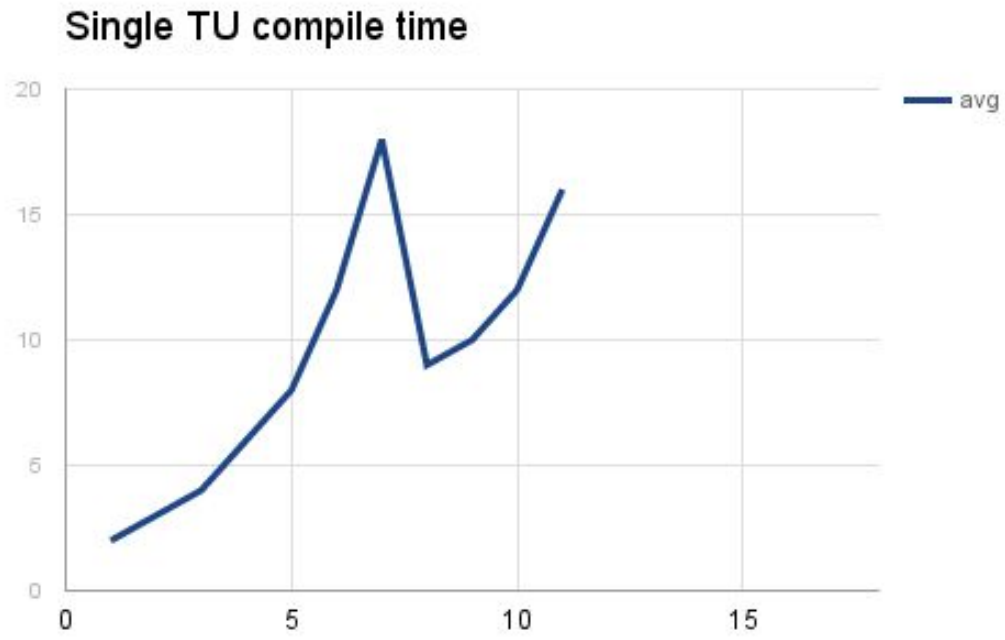
Flipping the switch

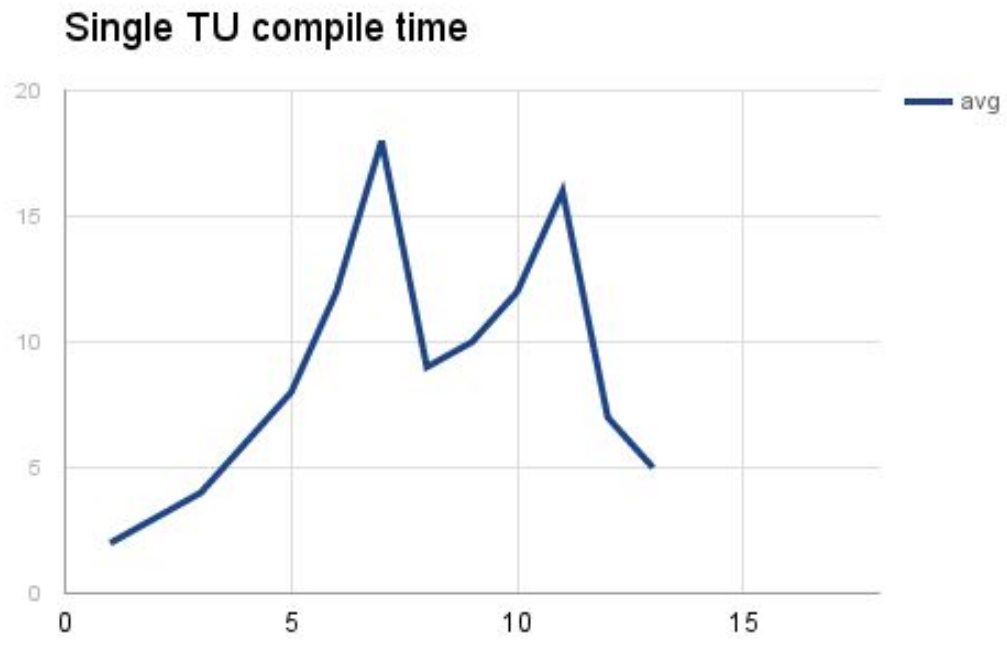
- headers are self-contained (> 10k changes)

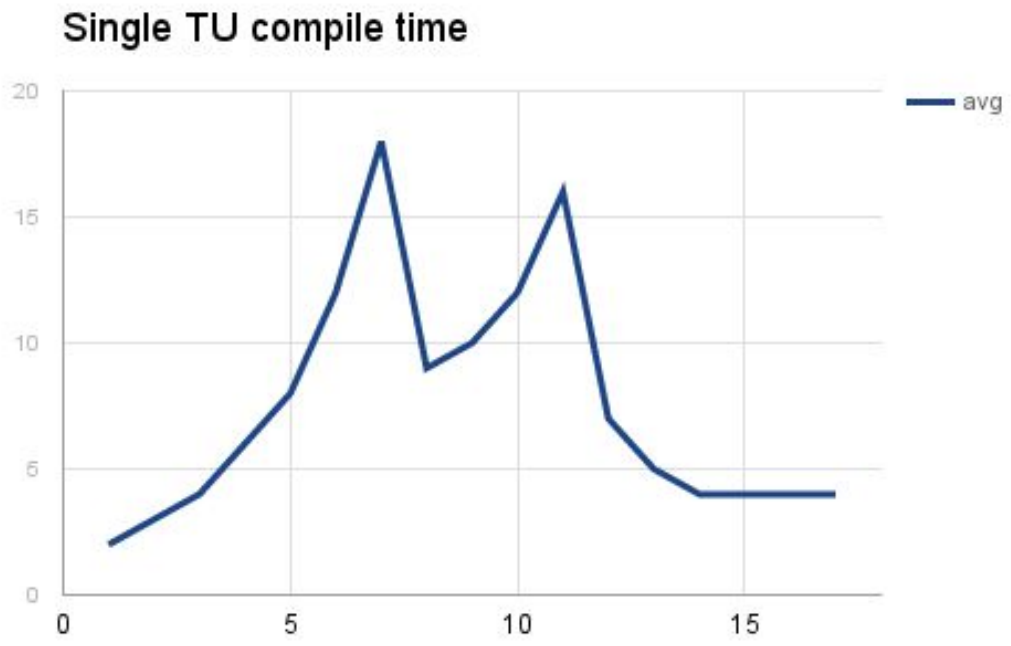
- configuration management is under control

- distributed build system supports modules

- flip the switch!


Single TU compile time
20
15
10
5
0
0
5
10
15
avg


Single TU compile time
20
15
10
5
0
0
5
10
15
avg

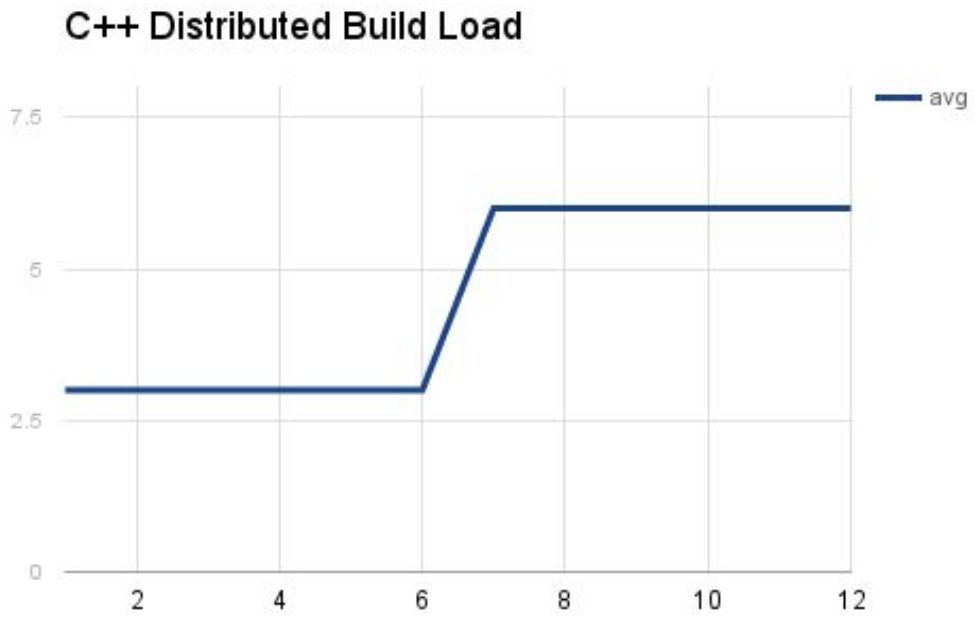

Single TU compile time
20
15
10
5
0
0
5
10
15
avg

:)

+ incremental rebuild better by up to 2x for the 99%ile

+ incremental rebuild average 10% better

The Big Regression

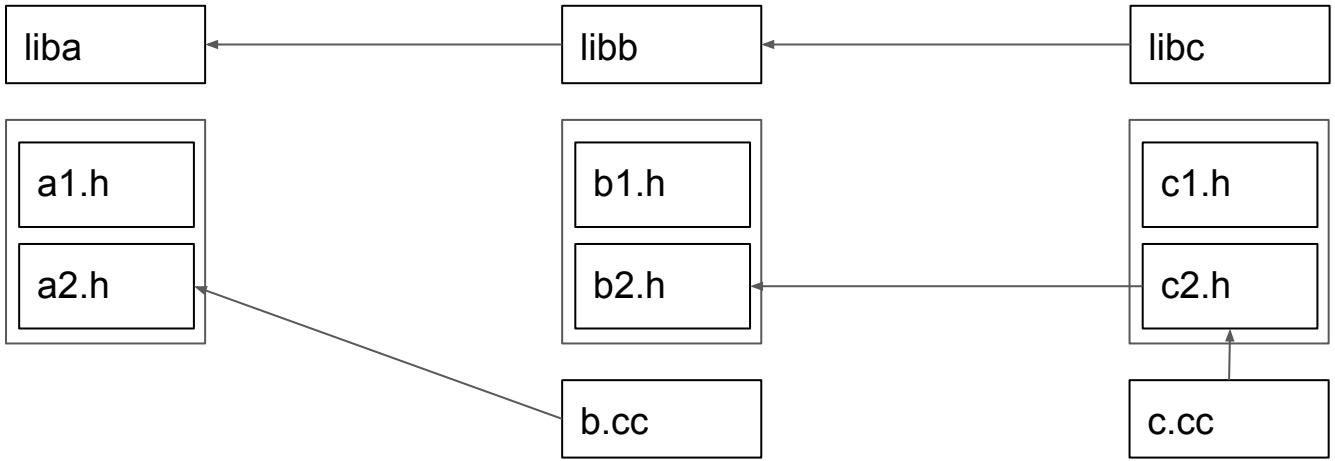


:(

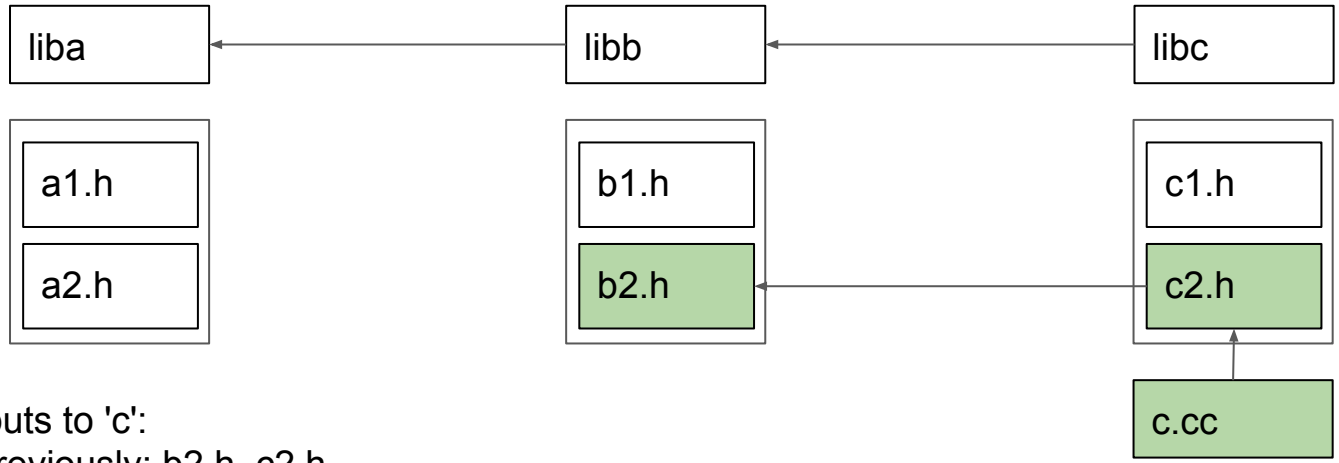
- compiling a lot more translation units

- overall load increased due to increase in #compiles

Building too much

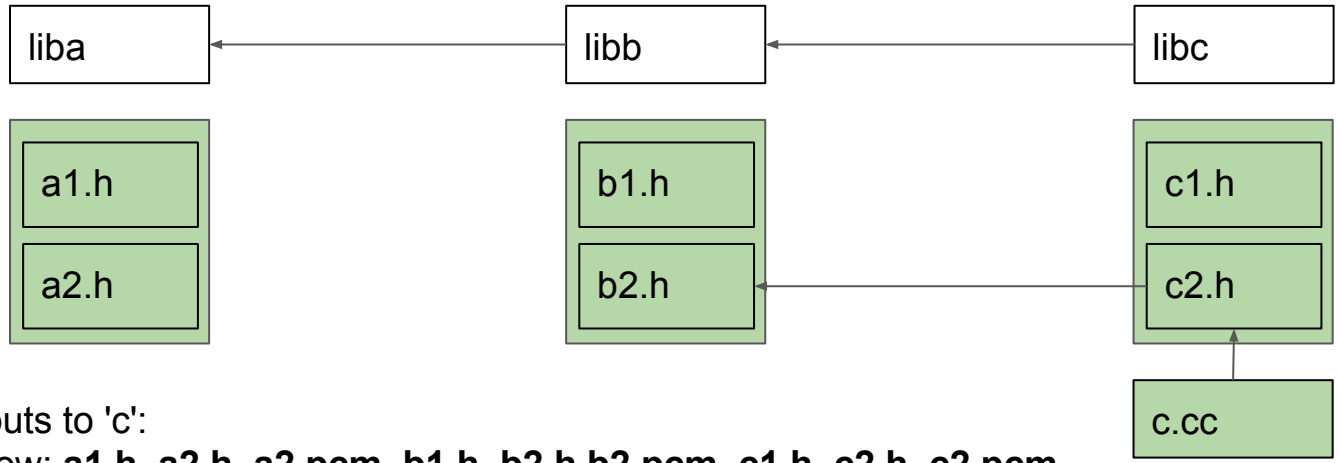


Building too much



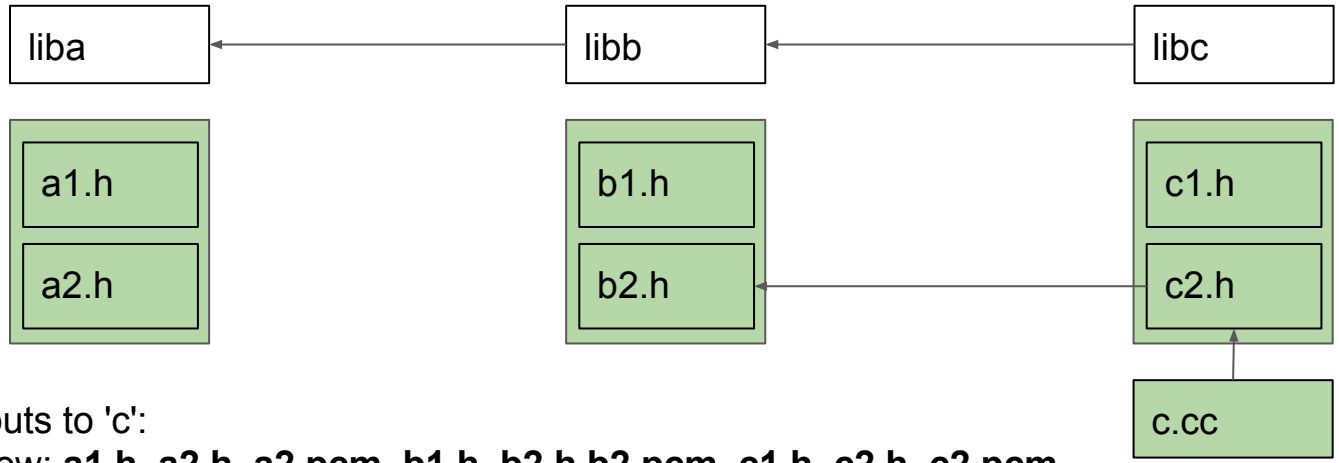
Inputs to 'c':
- previously: b2.h, c2.h

Building too much



Inputs to 'c':
- now: **a1.h, a2.h, a2.pcm, b1.h, b2.h b2.pcm, c1.h, c2.h, c2.pcm**

Building too much

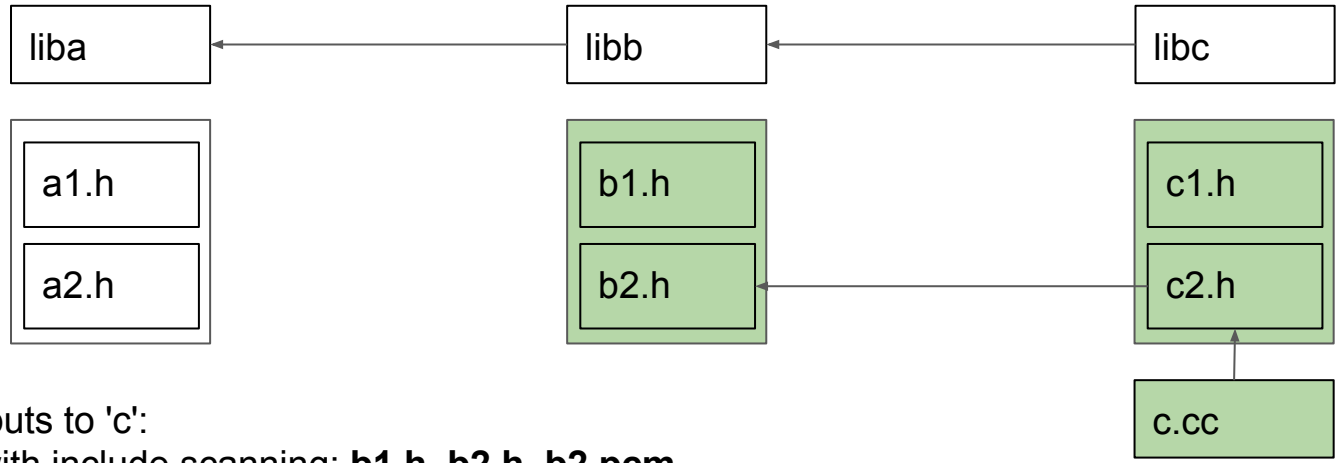


Inputs to 'c':
- now: **a1.h, a2.h, a2.pcm, b1.h, b2.h b2.pcm, c1.h, c2.h, c2.pcm**

- compile time **faster,** setup time **slower**

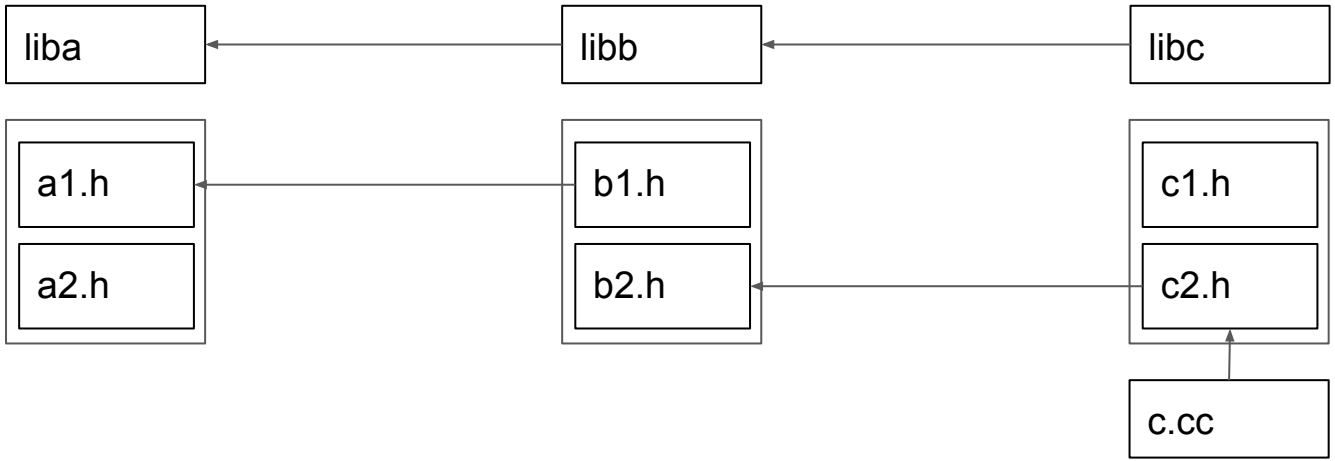
- lots more recompiles

Building too much

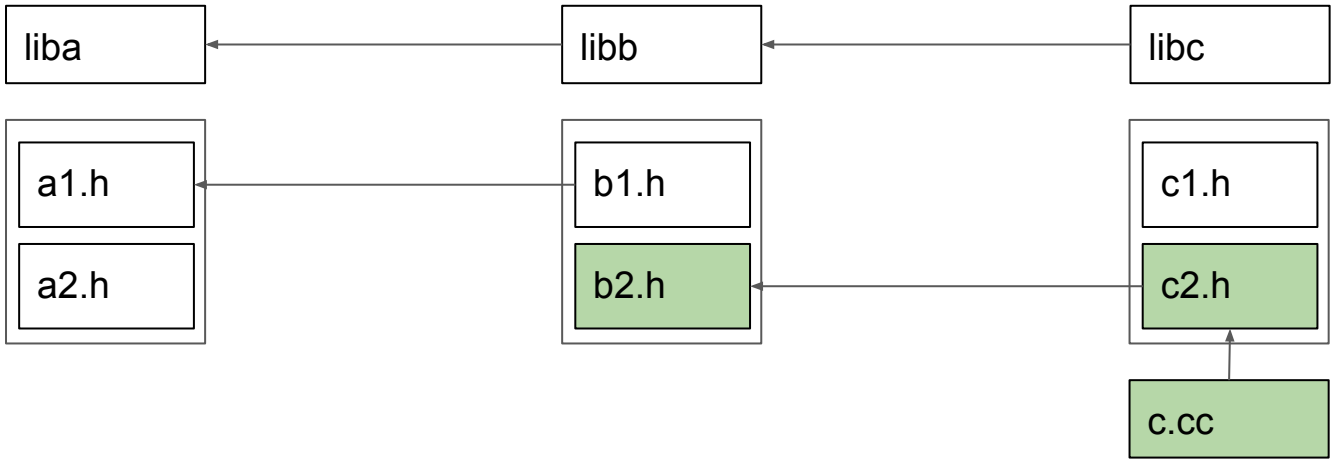


Inputs to 'c':
- with include scanning: **b1.h, b2.h, b2.pcm**

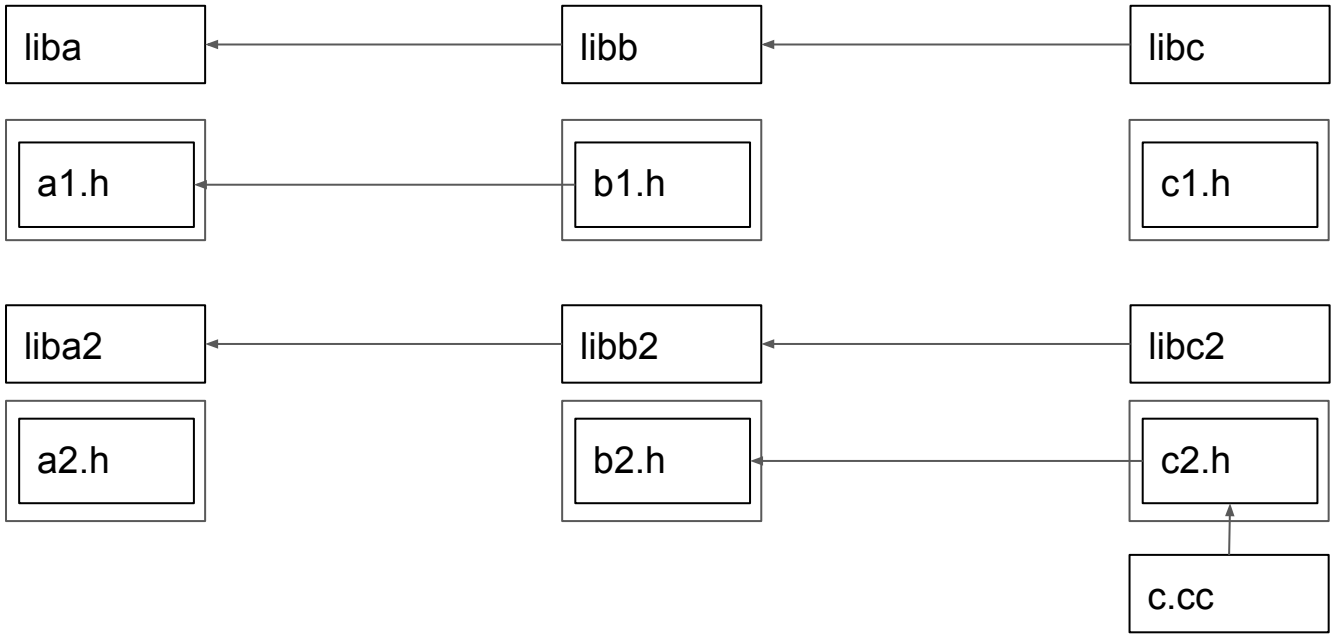
Building too much



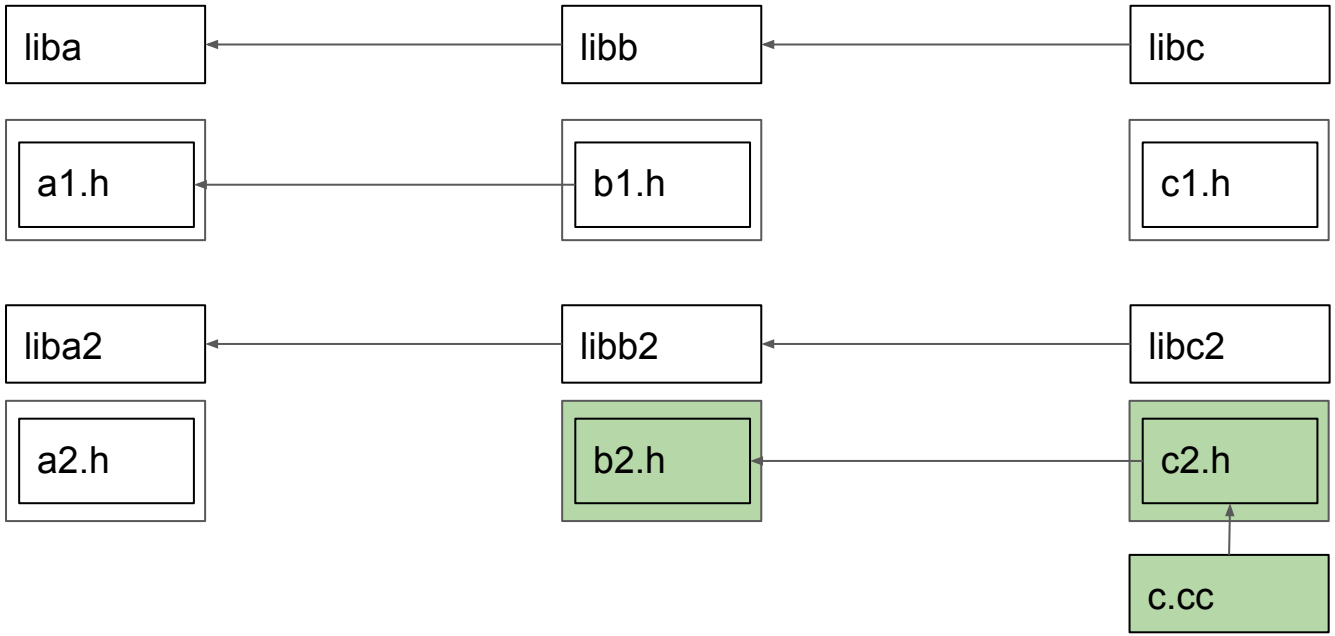
Building too much



Building too much



Building too much



Wrapping it up...

Results

- incremental compiles improved
    - up to 50% in the 99%-ile
    - avg ~10%

- clean build times ~equivalent

- overall build load increased due to increasing #compiles

- unlocked loads of optimization potential

- prepares the code base for C++ modules

What worked

- we were lucky: style guide -> most things pretty good

- tools to fix things at scale -> fix broken code

- build system models headers -> build system integration doable

What was problematic

- performance improvement not as large as expected

- broken code sometimes really hard to fix

- lots of effort tweaking the distributed build system

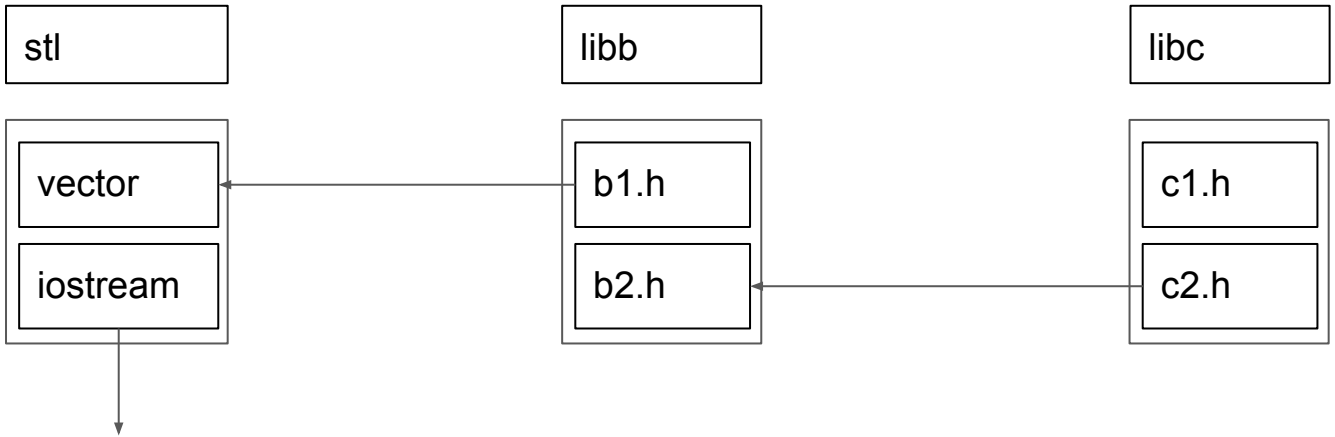
- bleeding edge of clang's features -> bugs

Questions?!

Rolling out Modules

1. Implement modules support in the build system
2. Make headers self-contained (-x c++-header)
3. Get configurations under control
4. Switch on modules
5. Fix things that break
6. Optimize your build system

Growing linker input




```
static ios_base::Init __ioinit;
```

The build system

```
proto_library(name="p", srcs=["p.proto"])

cc_library(name="a", hdrs=["a1.h", "a2.h"],
           srcs=["a.cc"])

cc_library(name="b", hdrs=["b.h"], srcs=["b.cc"],
           deps=["a", "p"])
```

b.cc:

```
#include "a1.h"
#include "b.h"
#include "other.h"
#include "p.pb.h"
```