Compilation Speedup Using C++ Modules:

A Case Study

CHUANQI XU

Alibaba Cloud

Compilation Speedup Using C++ Modules: A Case Study

Chuanqi Xu

Reply To: yedeng.yd@linux.alibaba.com

Menu

Alibaba Cloud

Introduction

Case Study

Possible Improvement

Menu

Alibaba Cloud

Introduction

- Brief introduction to modules
- The Complexity Theory
- The impact on compilation model
- Inline function and templates

Case Study

- The studied case
- The test environment
- The measuring method
- The experiment Results
- The analysis

Possible Improvement

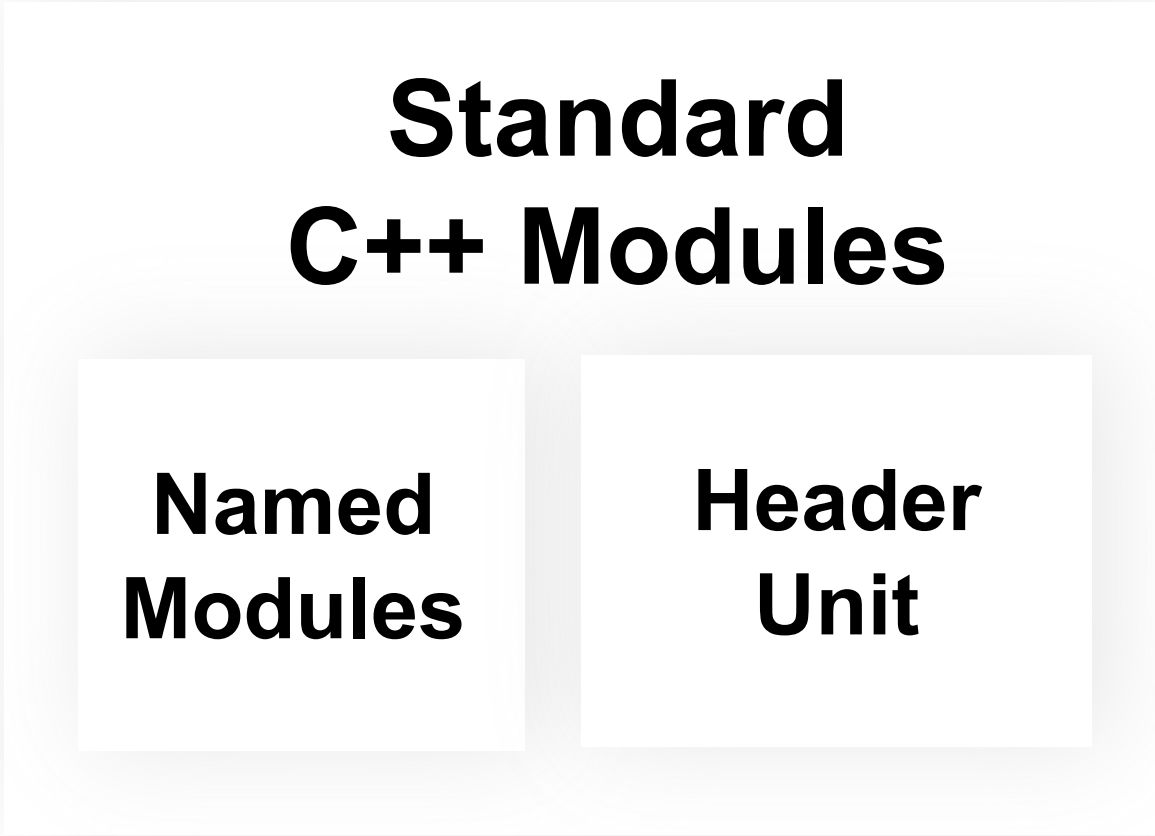
- Possible Improvement on the compiler side
- Possible Improvement on the library side
- The improvement in the longer term

Introduction

Alibaba Cloud

What are Modules

- Not strictly, Modules can be divided into Named Modules and Header Units.
- Named Modules can be compiled into object files while Header Units can't.
- The study talks about Named Modules Only.



Note: Technically, Header Units are synthesized translation units. The declarations within Header Units are attached to global module. Global Module has no name… but let's not be pedantic in this talk.

Introduction

Alibaba Cloud

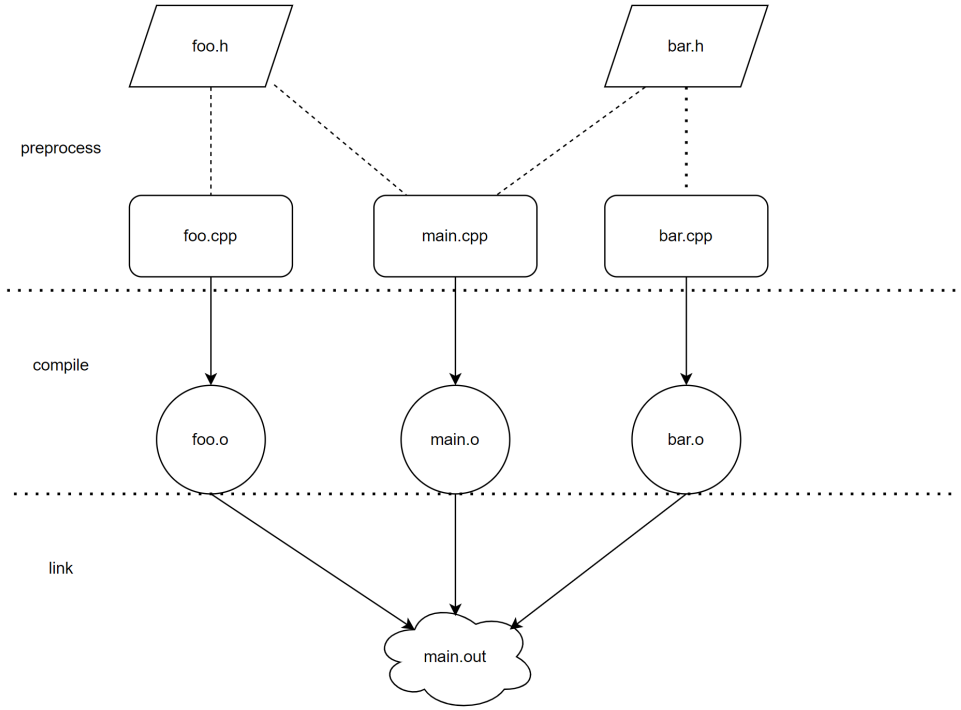
What are Modules

- With modules, we can `import` our dependencies instead of `#include` them.
- A module can consist of multiple module units.
- With modules, the project organization could be different with the previous style with headers.

```
1    import std;
2    int main() {
3        std::printf("Hello World.\n");
4    }
```

Introduction

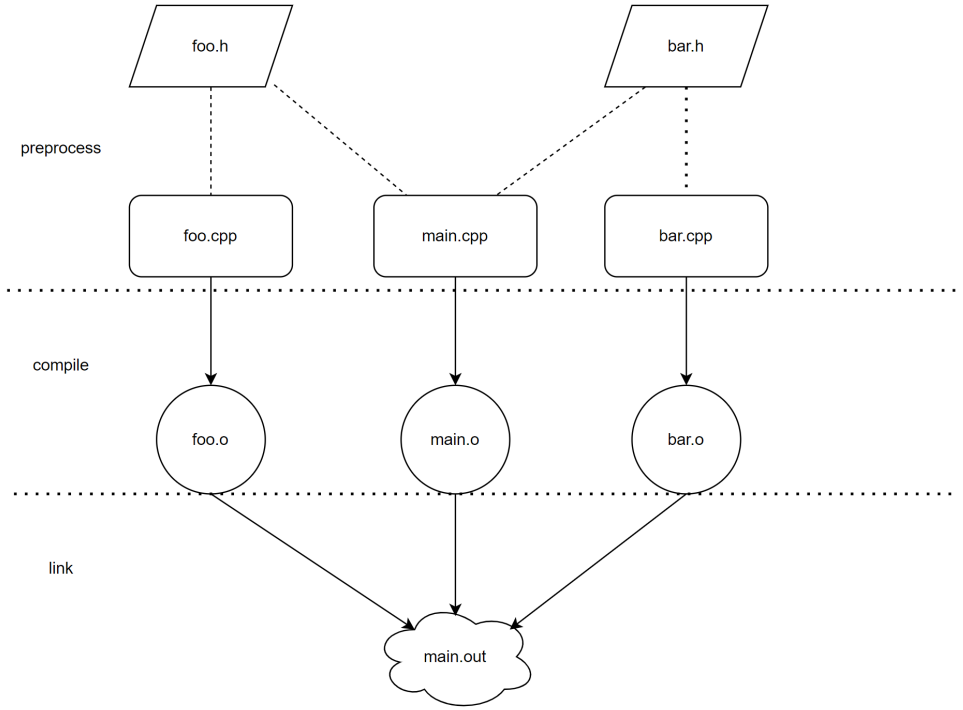
Example

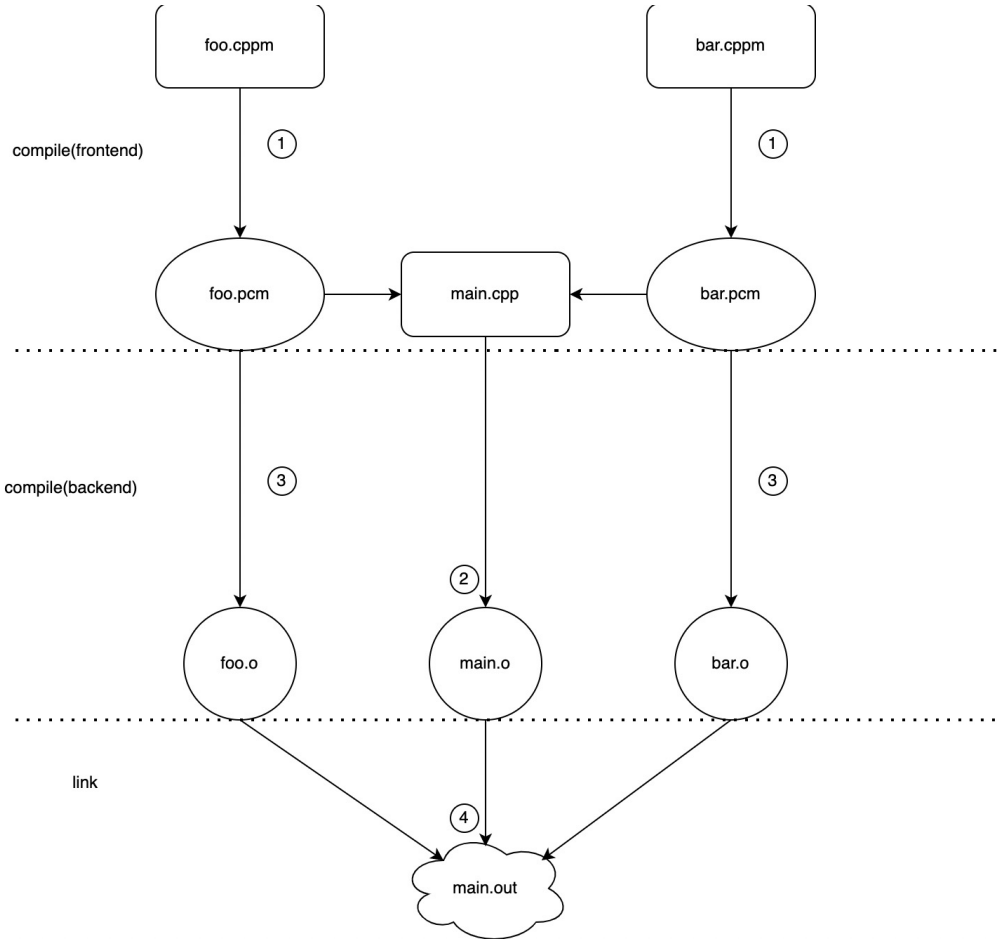


- `foo.h` and `bar.h` declare some declarations and `foo.cpp` and `bar.cpp` implement some declarations.
- The codes in `foo.h` and `bar.h` will be compiled twice.

Introduction

Example

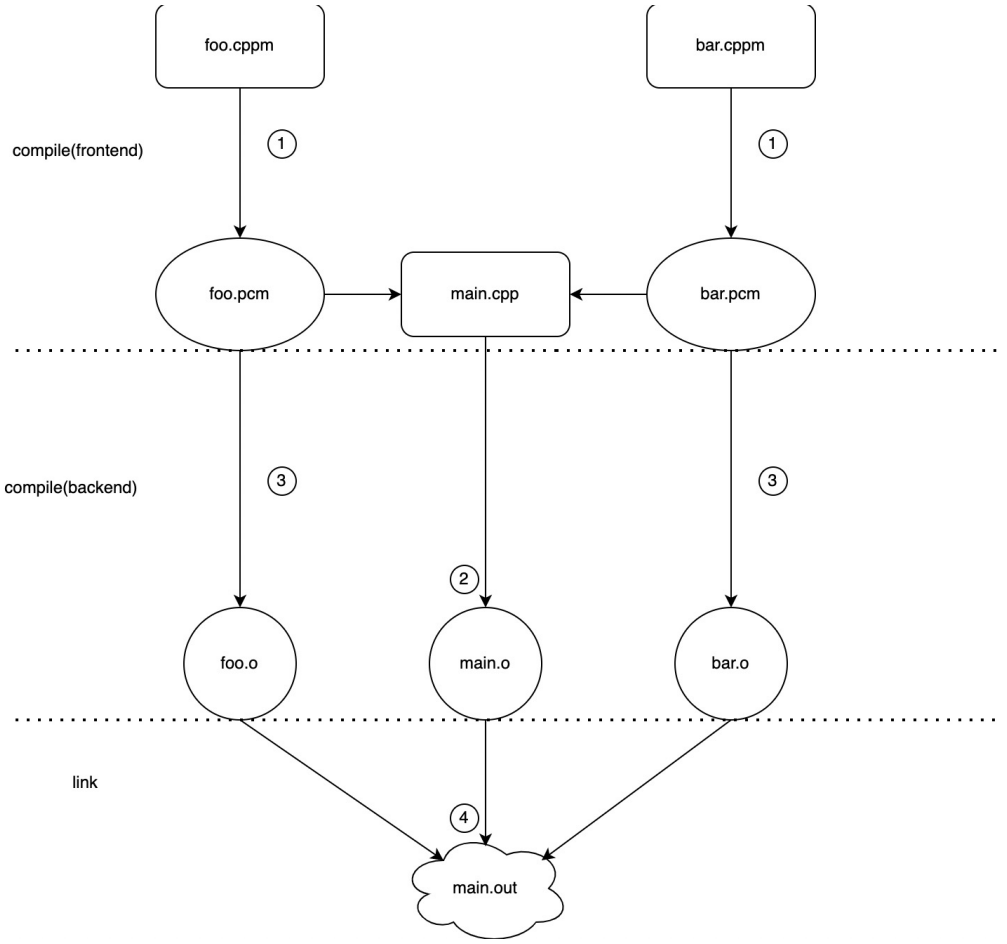




Introduction

Example

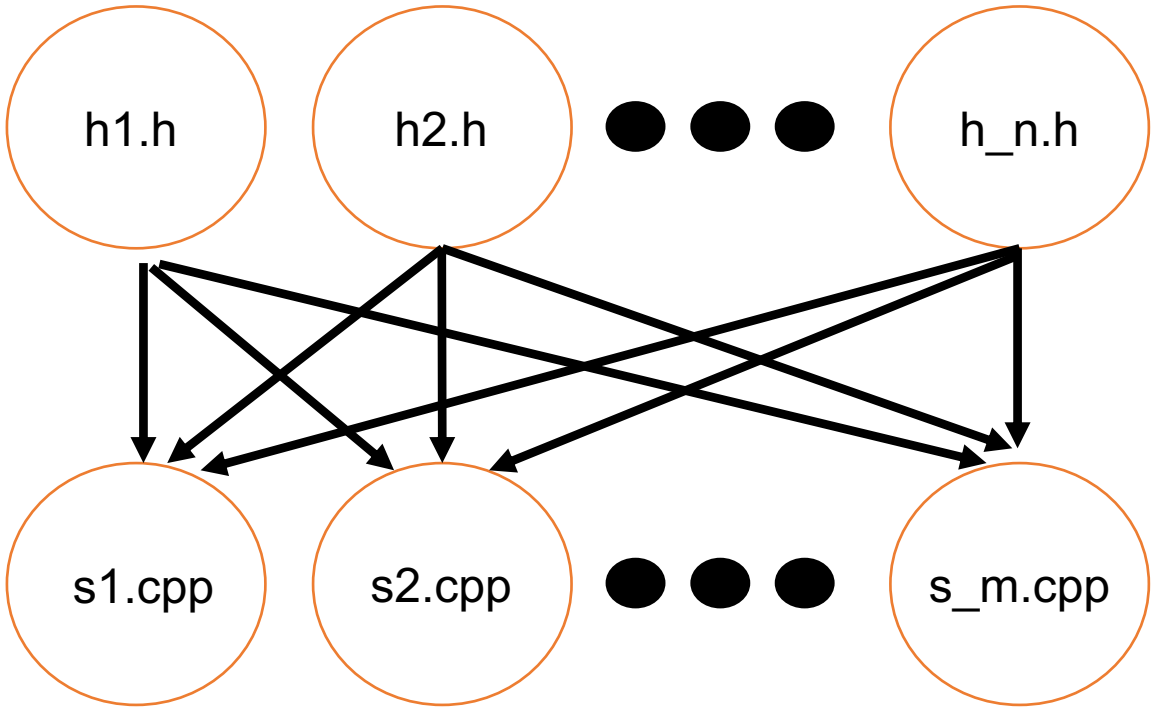
- Combine `foo.h` and `foo.cpp` into `foo.cppm`. So does `bar.cppm`.
- We need to compile `*.cppm` into prebuilt module interface so that the other unit can import them.
- Now the codes in `foo.cppm` and `bar.cppm` will be compiled once.



Introduction

Why are modules faster?

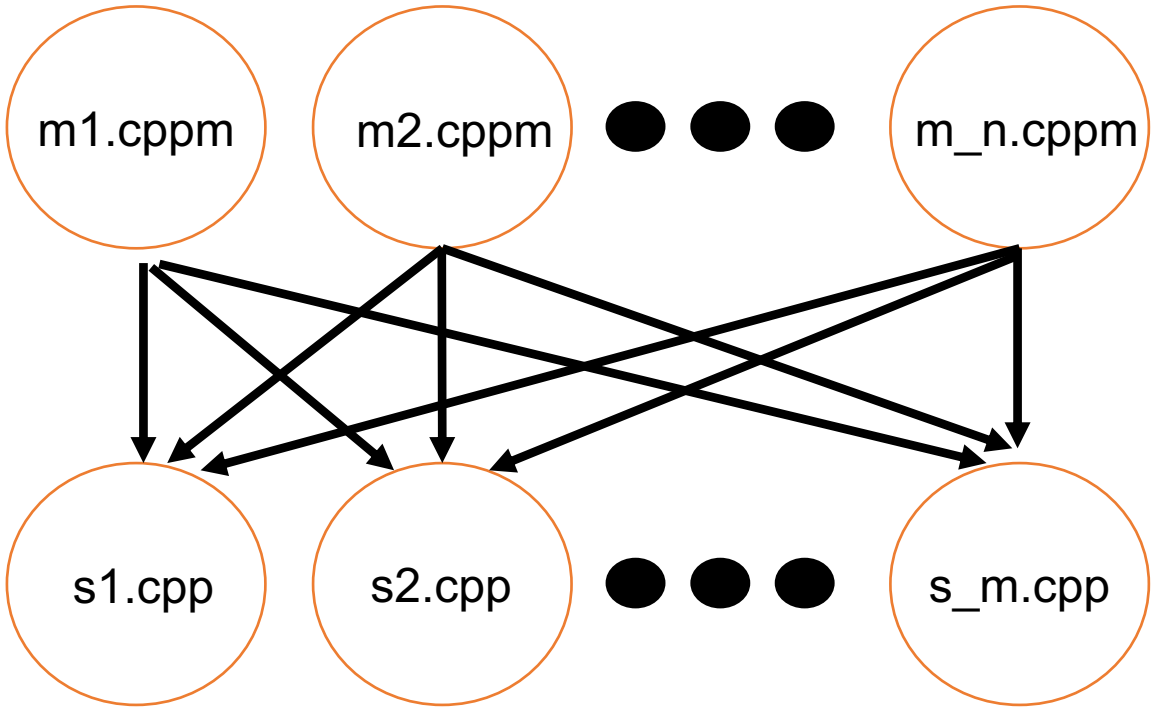
- In an ideal model, there are `n` headers and `m` sources. And every source requires every headers.
- Given each header and each source (without header) require the same time to compile.
- So complexity will be `O(n*m)`.



Introduction

Why are modules faster?

- Similarly, if there are `n` module units and `m` sources and each source requires each module.
- And each module unit and each source (without header) require the same time to compile.
- The complexity will be `O(n+m)`.

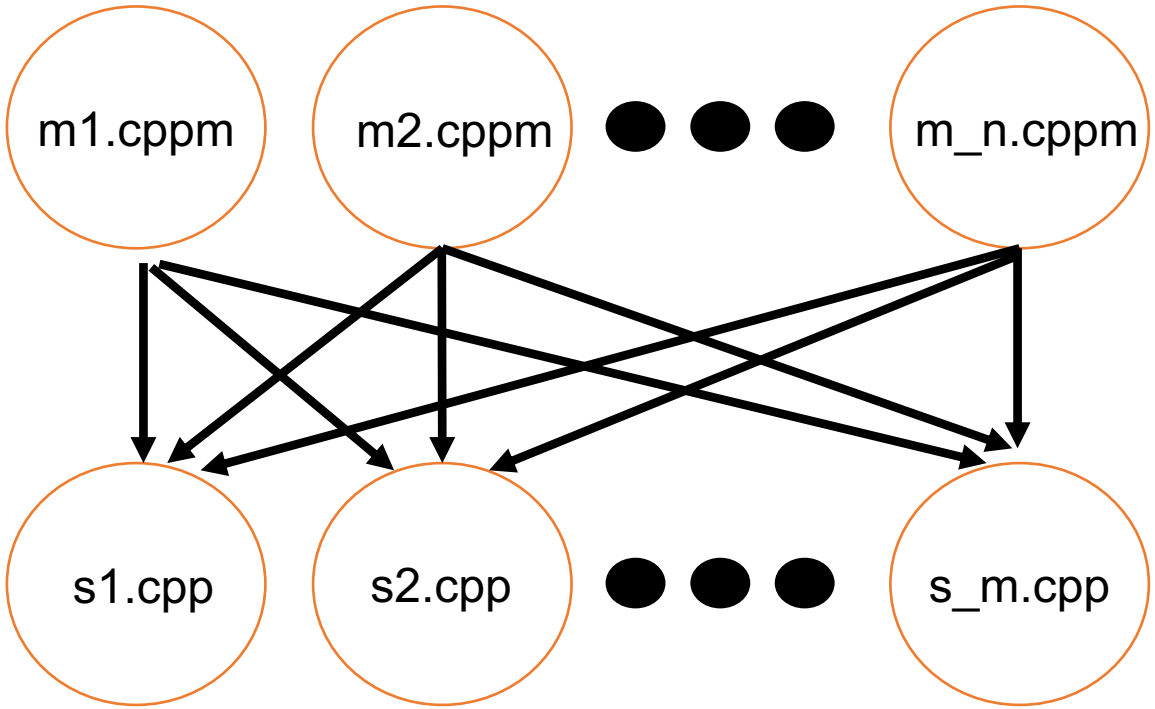


Introduction

Alibaba Cloud

Drawbacks

- The complexity theory is correct. But it is too rough so that the user could only feel it when the change scale are large enough.
- It ignores important features: compiler optimizations, inline functions and templates…

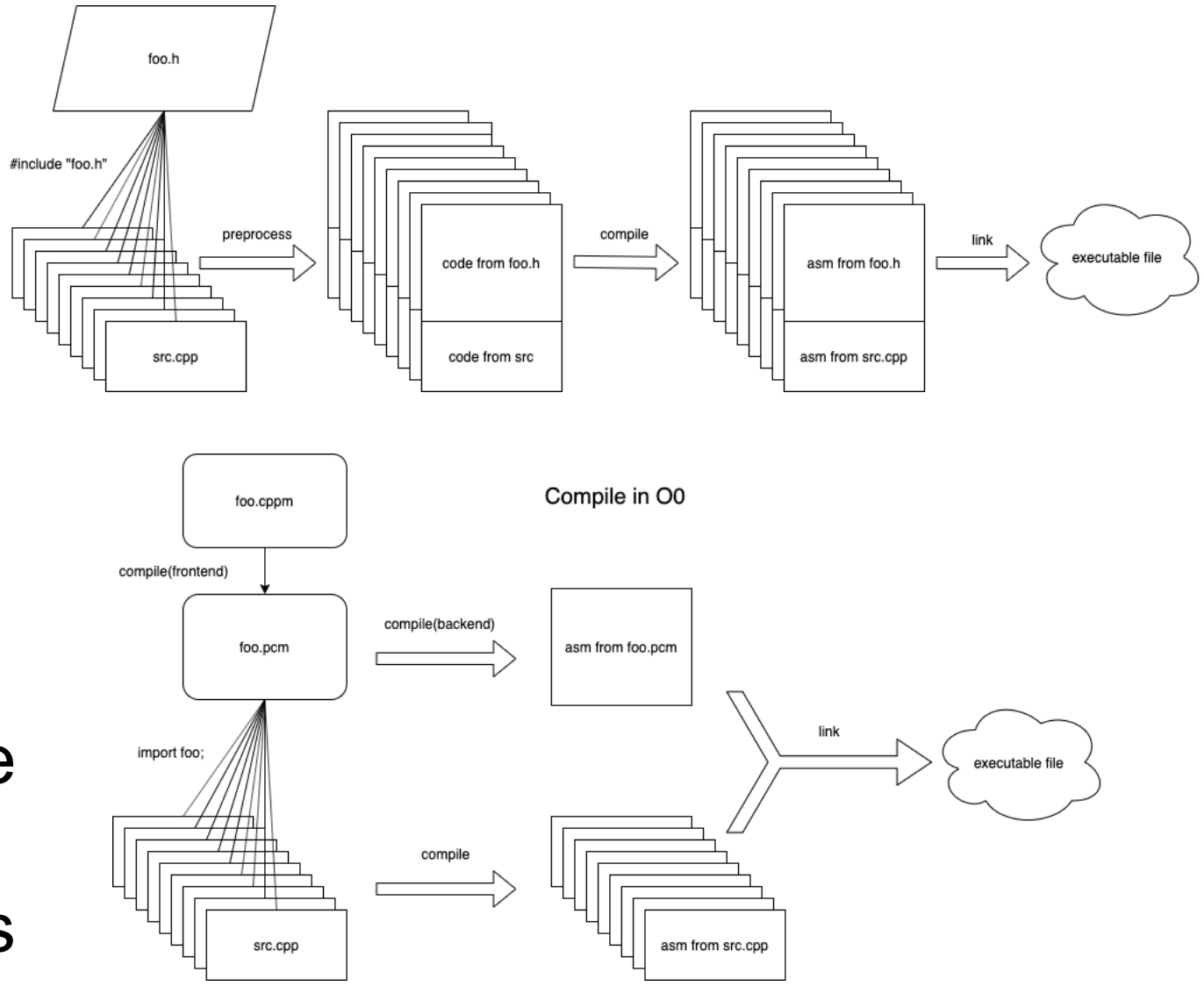


Note: In this talk, when we use `inline` as an adjective, we mean the `inline` semantic in the language side. When we use `inline` as a verb, we mean the compiler optimization technique `inlining`.

Introduction

Optimization Impact

- Another example to show the difference compilation process: we have only one header or one module unit.
- In the header example, the codes in foo.h will be compiled many times.
- In the modules example, the codes in foo.cppm will be compiled only once in O0 as expectedly.



Introduction

Alibaba Cloud

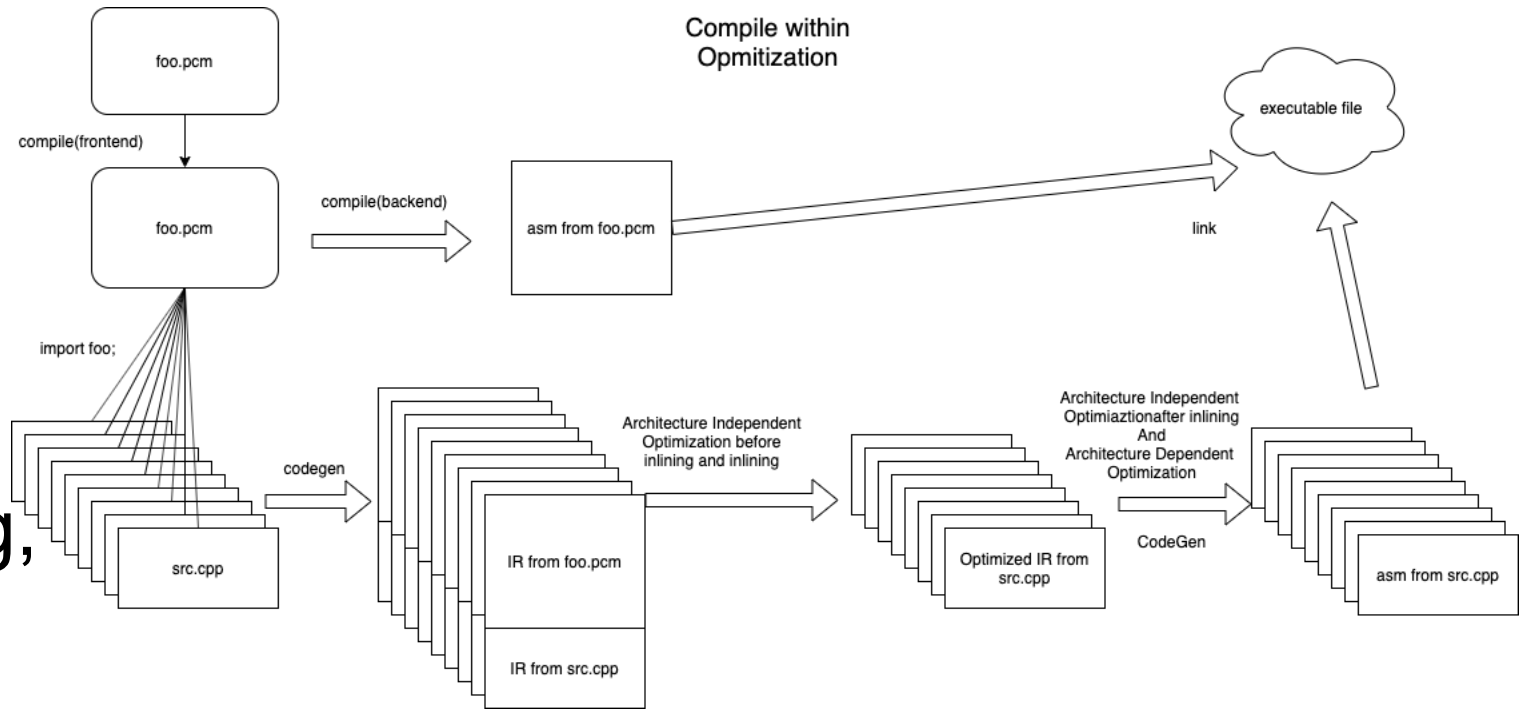
Optimization Impact

- However, it is different with optimizations.
- With optimizations, the compiler need to see function body as much as possible to perform IPO. (inter procedure optimizations, primarily inlining)
- It is unacceptable to hurt the runtime performance for compile time performance.
- So when we import a module within optimization, the called function bodies are imported and the compiler will keep them until inlining.

Introduction

Optimization Impact

- As the diagram shows, within optimizations, we can still save the time for optimizations after inlining, backend optimization and code generation.
- However, the optimizations before inlining and inlining itself take a lot of time.



So the speedup within optimization is significantly less than O0.

Introduction

Alibaba Cloud

Inline function Impact

The inline functions' bodies will be imported too even if in O0 according to the requirement of ABI.

In the following example, we can find the function body of foo in generated code for use.cpp

```
// foo.cppm
export module foo;
export inline int foo() {
    return 43;
}

// use.cpp
import foo;
int use() {
    return foo();
}
```

```
define dso_local noundef i32 @_Z3usev() #0 {
entry:
  %call = call noundef i32 @_ZW3foo3foov()
  ret i32 %call
}

define linkonce_odr dso_local noundef i32 @_ZW3foo3foov() #1 comdat {
entry:
  ret i32 43
}
```

Introduction

Alibaba Cloud

Templates Impact

- We said `The codes in module…` many times. It is anti-intuitive if we write template in the module.
- The templates will be instantiated only if they are used.
- Naturally, if they are used in the importee, the generated code lives in the importee too.

```
// foo.cppm
export module foo;
export template <class T>
int foo() {
    return 43;
}

// use.cpp
import foo;
int use() {
    return foo<int>();
}
```

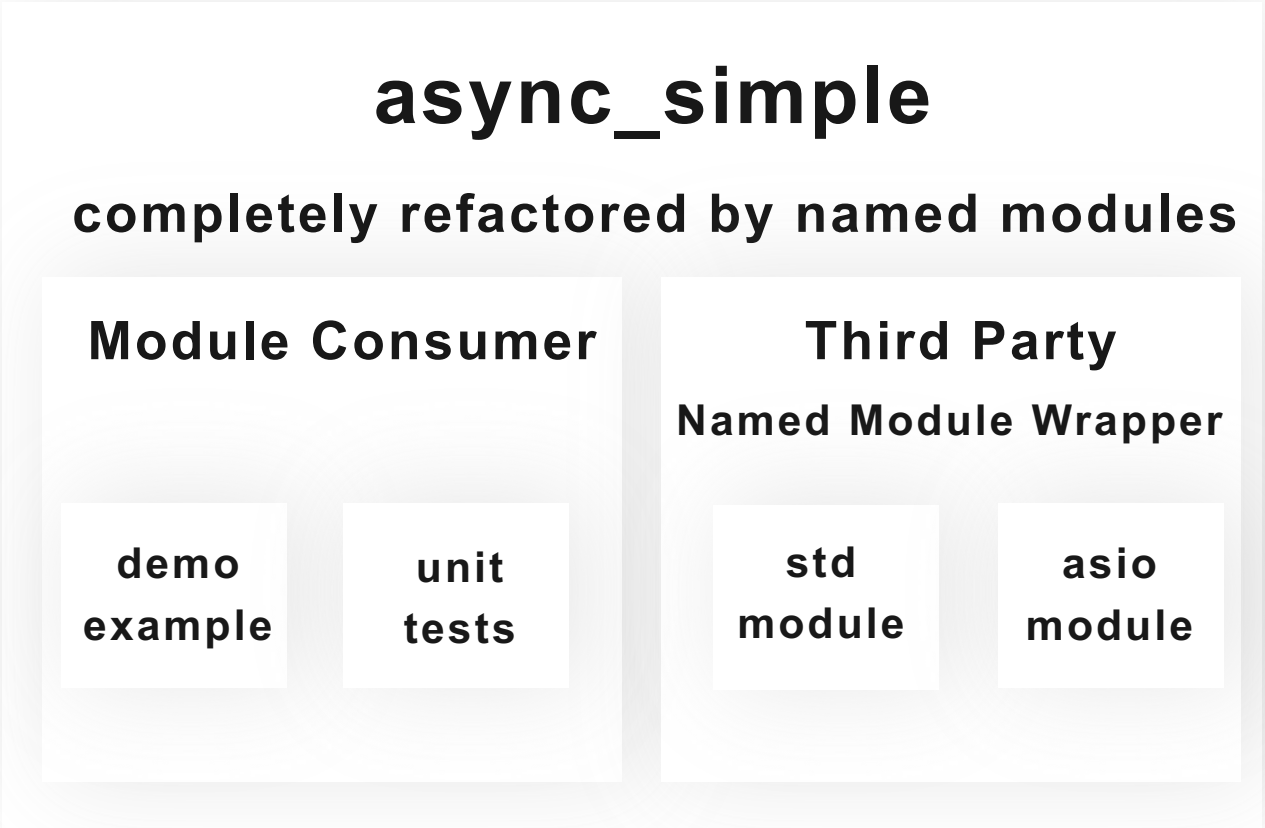
The generated code for `foo.cppm` contains nothing. All the codes lives in use.cpp

Case Study

Alibaba Cloud

Project Structure

- The library `async_simple` itself is refactored into a named module completely.
- It wraps two third party libraries (std and asio) into named modules.
- It contains two module consumers. The demo example uses modules completely. The unit tests include gtest since gtest depend on macro. It provides diversity.



The studied case lives in: https://github.com/alibaba/async_simple/tree/CXX20Modules

Case Study

Quick Profiling

Alibaba Cloud

	#Files	Lines Of Codes
async_simple	31	~3,300
unit tests	10	~2,500
demo example	13	~1,300
asio	572	~92,000
std	omitted	omitted

Case Study

Quick Profiling

Modules	Where are used	What are used
async_simple	tests and demo example	Almost templates: Lazy<T>, Future<T>, ···
asio	demo example	Nearly No templates: socket, io_context, ···
std	async_simple, tests and demo	A lot of templates: vector<T>, ···

Case Study

Alibaba Cloud

Testing Enviroment

- Compiler: a downstream implementation for clang13.
- STD Library: libstdc++10.3
- Build System: Hand written Makefile
- Operating System: CentOS 4.9
- Hardware: Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz; cpu MHz : 2699.584 processor number: 96; CPU Caches: L1 Data 32 KiB (x48) L1 Instruction 32 KiB (x48) L2 Unified 1024 KiB (x48) L3 Unified 33792 KiB (x2)

Case Study

Alibaba Cloud

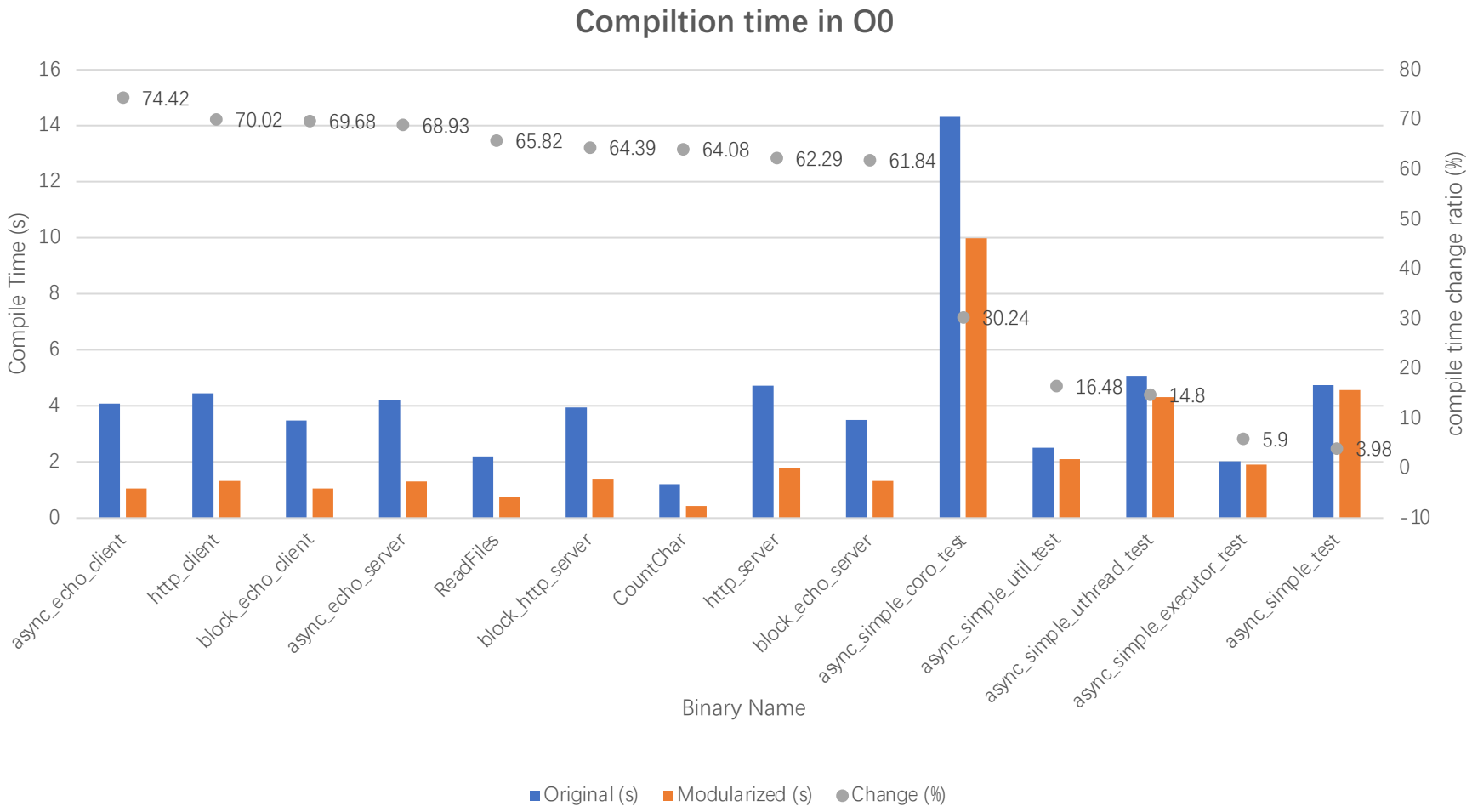
Measuring Method

- What do we measure: we measure the compilation time for demo example and tests for modular build and the original build. Since the original async_simple is a nearly header only library, it takes no time to build itself. So we don’t compare it.
- How do we measure: We measure them with different optimization levels. We always build them with `-g` enabled since we enabled `-g` all the time. And the processing of debug information should be nearly a ratio constant. e.g., whether or not to enable `-g` shouldn’t affect the difference ratio drastically.

Case Study

Results

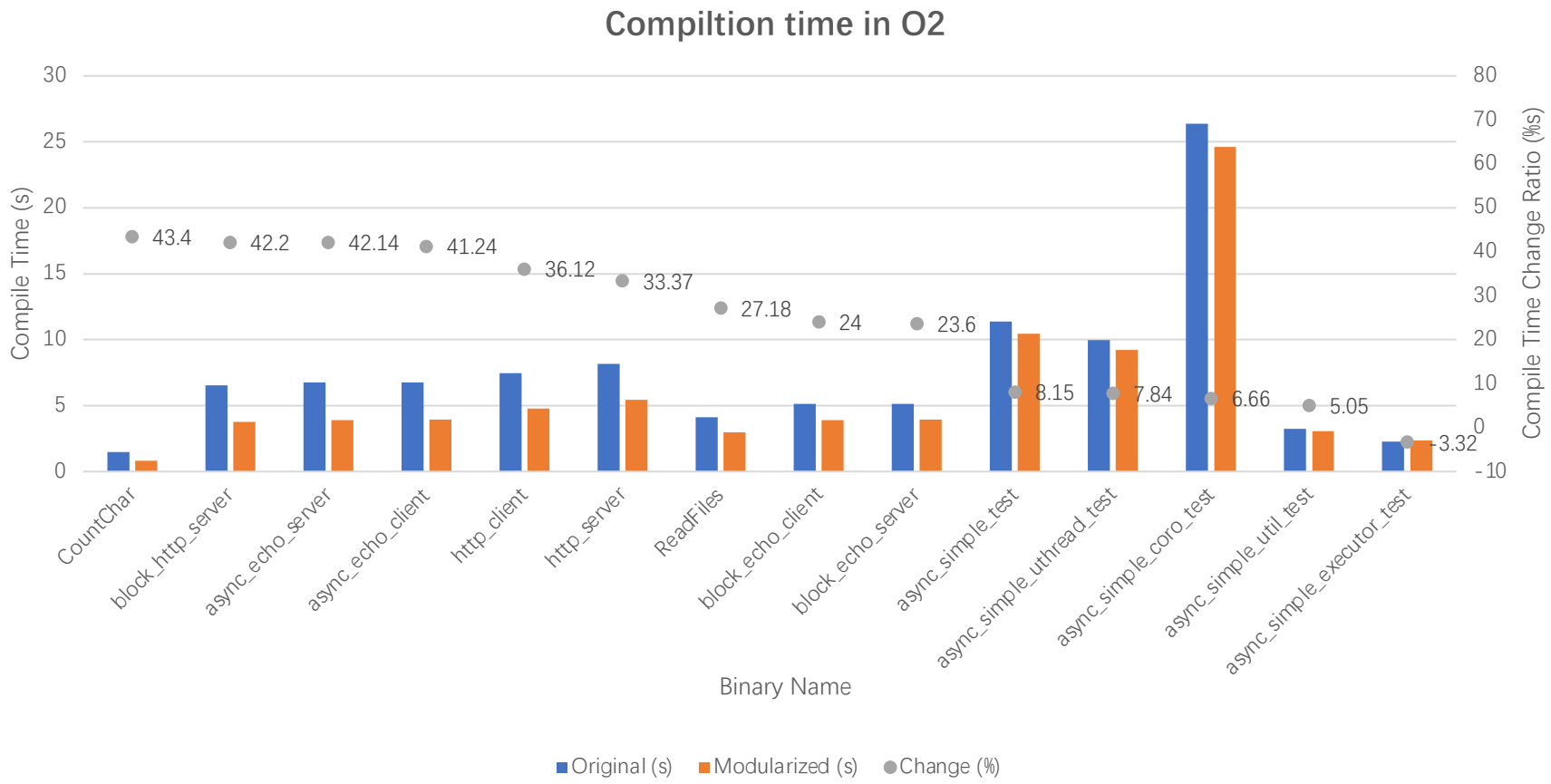
- Everything looks fine in O0.
- The highest speedup is nearly 4.x.
- The average speedup for demo example is 2.32x.
- But for tests, we get much less improvement.



Case Study

Results

- We get much less speedup than O0.
- The highest speedup is 1.7x.
- And the average speed for demo examples is 1.28x.
- The results for tests are much worse. None of tests save time more than 10%.



Case Study

Alibaba Cloud

The analysis – why demo have higher speedup

```
16  import std;
17  import asio_util;
18
19  using asio::ip::tcp;
20
```

Figure1: how demo depents

It can use named modules completely

```
16  #include <gmock/gmock.h>
17  #include <gtest/gtest.h>
18
19  import async_simple;
20  import std;
```

Figure2: how test depents

The use of gtest relies on macros heavily. So we need to use headers. Then the contents of the headers is part of the TU.

```
16  export module asio_util;
17  import std;
18  export import async_simple;
19  export import asio;
```

ps: the implementation of asio_util.

Case Study

Alibaba Cloud

The analysis – why demo have higher speedup

- Demo uses named modules completely while tests use headers, too.
- While async_simple provides primarily templates, the demo uses mostly non template components from asio.

Case Study

Alibaba Cloud

The analysis – why demo have higher speedup

- Demo uses named modules completely while tests use headers, too.
- While async_simple provides primarily templates, the demo uses mostly non template components from asio.

Conclusion 1: the speedup from modules depends on the pattern and the usage.

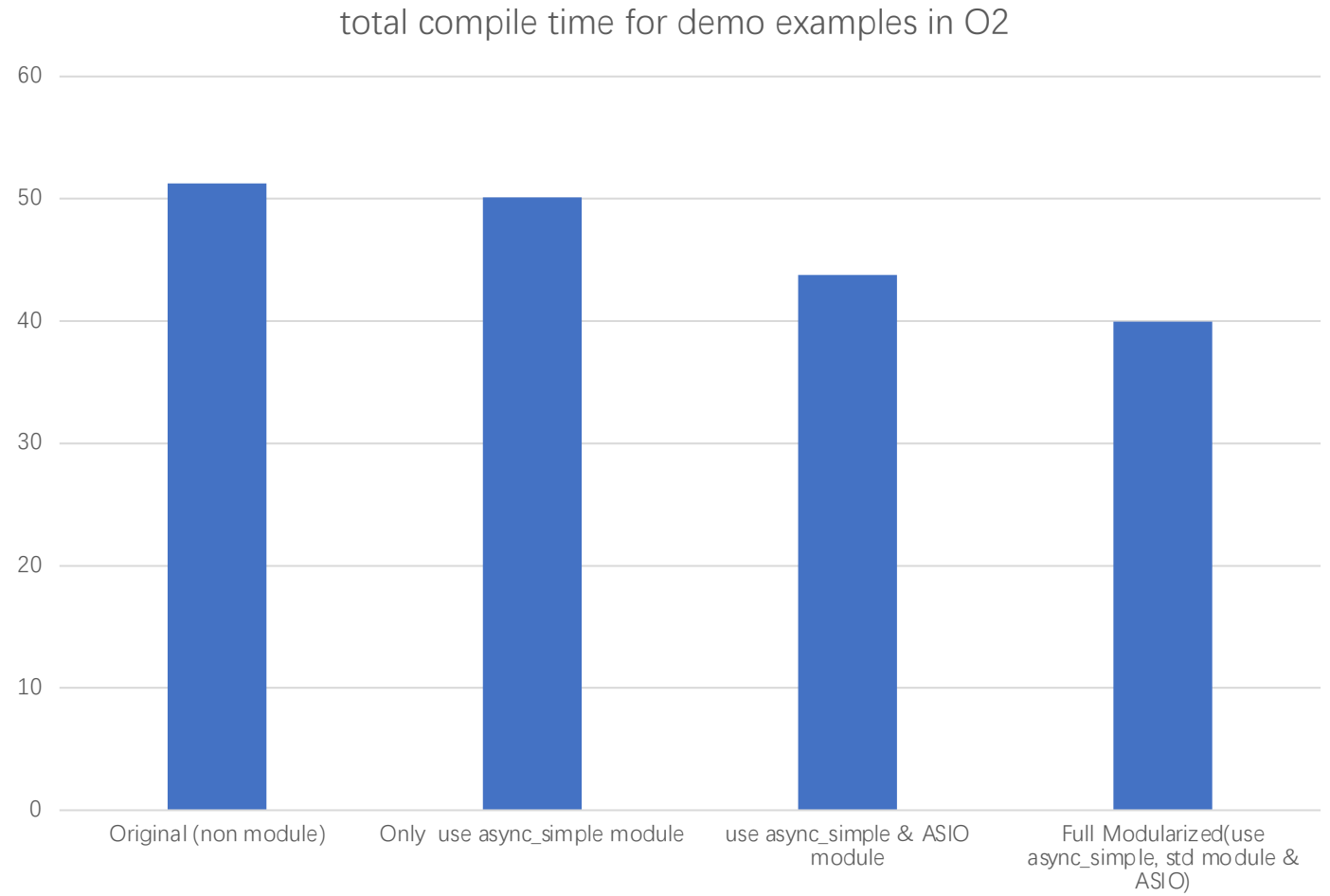
Case Study

Alibaba Cloud

The analysis – how different modules affect the speed

We measured the compilation time change for demo examples with different modules used.

Here we can observe that we get the biggest time decreasement after we use asio module.



Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

- Motivation 1: why we get much less speedup within optimizations? Since the optimizer needs to see the function body from other module unit.

Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

- Motivation 1: why we get much less speedup within optimizations? Since the optimizer needs to see the function body from other module unit.
- Motivation 2: there is a compiler optimization technique called LTO (or ThinLTO) which can combine different TU together to optimize.

Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

- Motivation 1: why we get much less speedup within optimizations? Since the optimizer needs to see the function body from other module unit.
- Motivation 2: there is a compiler optimization technique called LTO (or ThinLTO) which can combine different TU together to optimize.
- Idea: When we enable LTO, if we can refuse to import function body from other module unit? Will we get performance loss? Can we control the performance loss?

Case Study - Possible Improvement

Alibaba Cloud

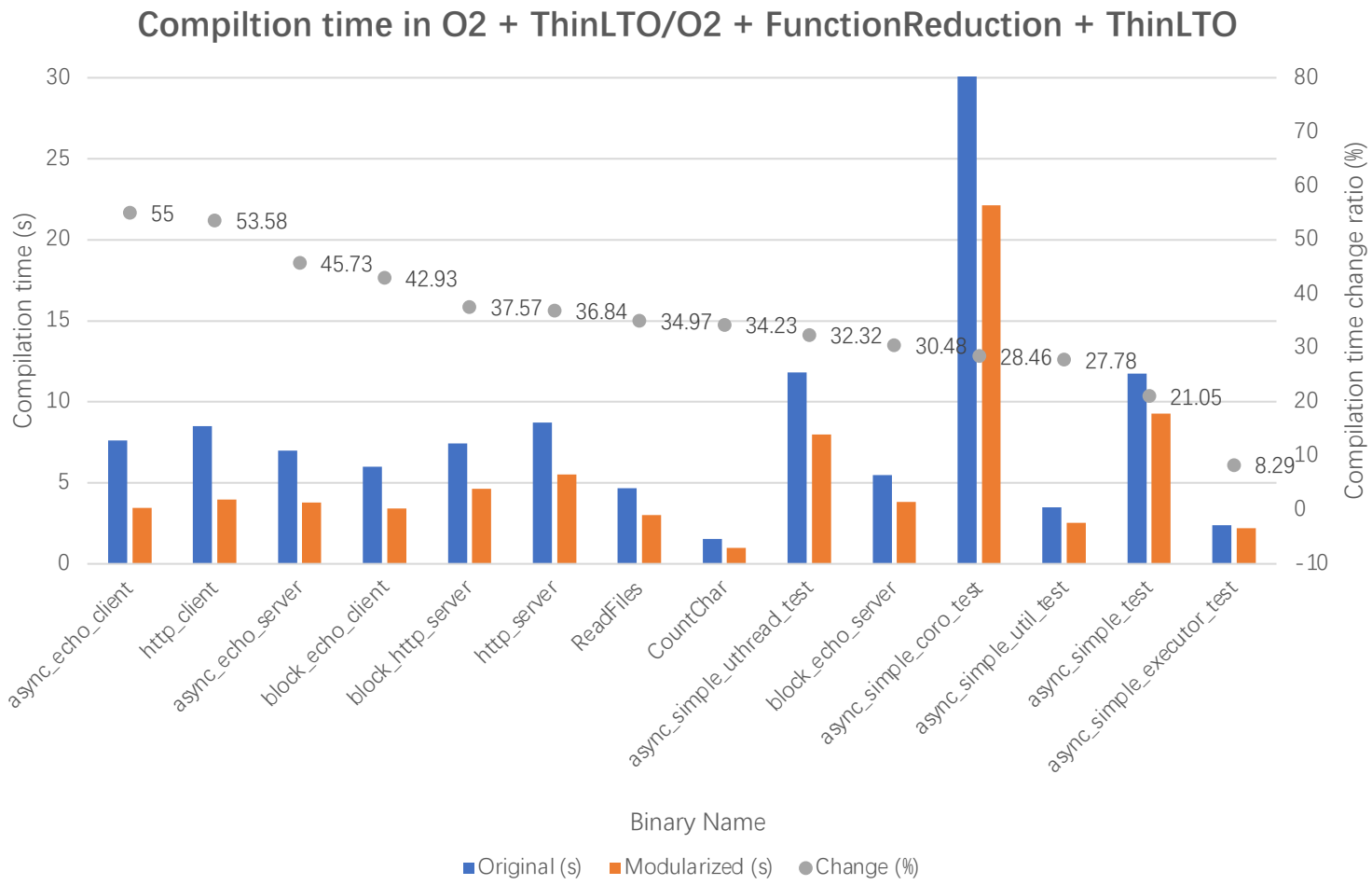
Compiler Improvement – Function Reduction

- Motivation 1: why we get much less speedup within optimizations? Since the optimizer needs to see the function body from other module unit.
- Motivation 2: there is a compiler optimization technique called LTO (or ThinLTO) which can combine different TU together to optimize.
- Idea: When we enable LTO, if we can refuse to import function body from other module unit? Will we get performance loss? Can we control the performance loss?
- We call this behavior as Function Redution and implemented it with our downstream compiler.

Case Study - Possible Improvement

Compiler Improvement – Function Reduction

Now we get much better compilation speedup within optimization.



Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

But how about performance?
How many people want
faster compilation with
worse performance?

Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

Background: compiler offers many options to control the behavior of ThinLTO. One is `-import-instr-limit`, which controls the threshold of lines of instructions the compiler would love to import. The default value for `-import-instr-limit` is 100.

Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

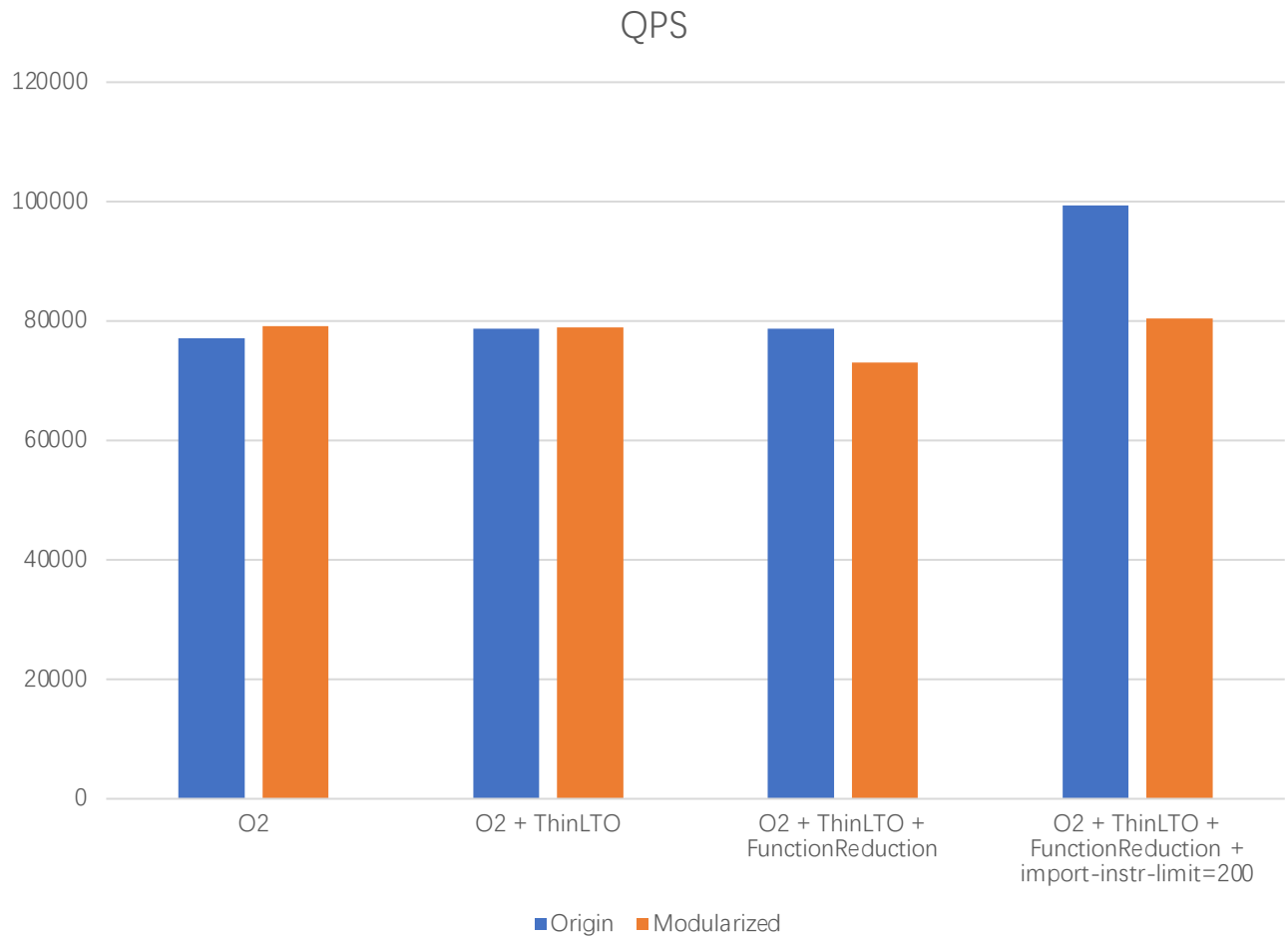
Background: compiler offers many options to control the behavior of ThinLTO. One is `-import-instr-limit`, which controls the threshold of lines of instructions the compiler would love to import. The default value for `-import-instr-limit` is 100.

Given Function Reduction prunes a lot of functions at the frontend. It makes sense to be more aggressive when importing function in ThinLTO.

Case Study - Possible Improvement

Compiler Improvement – Function Reduction

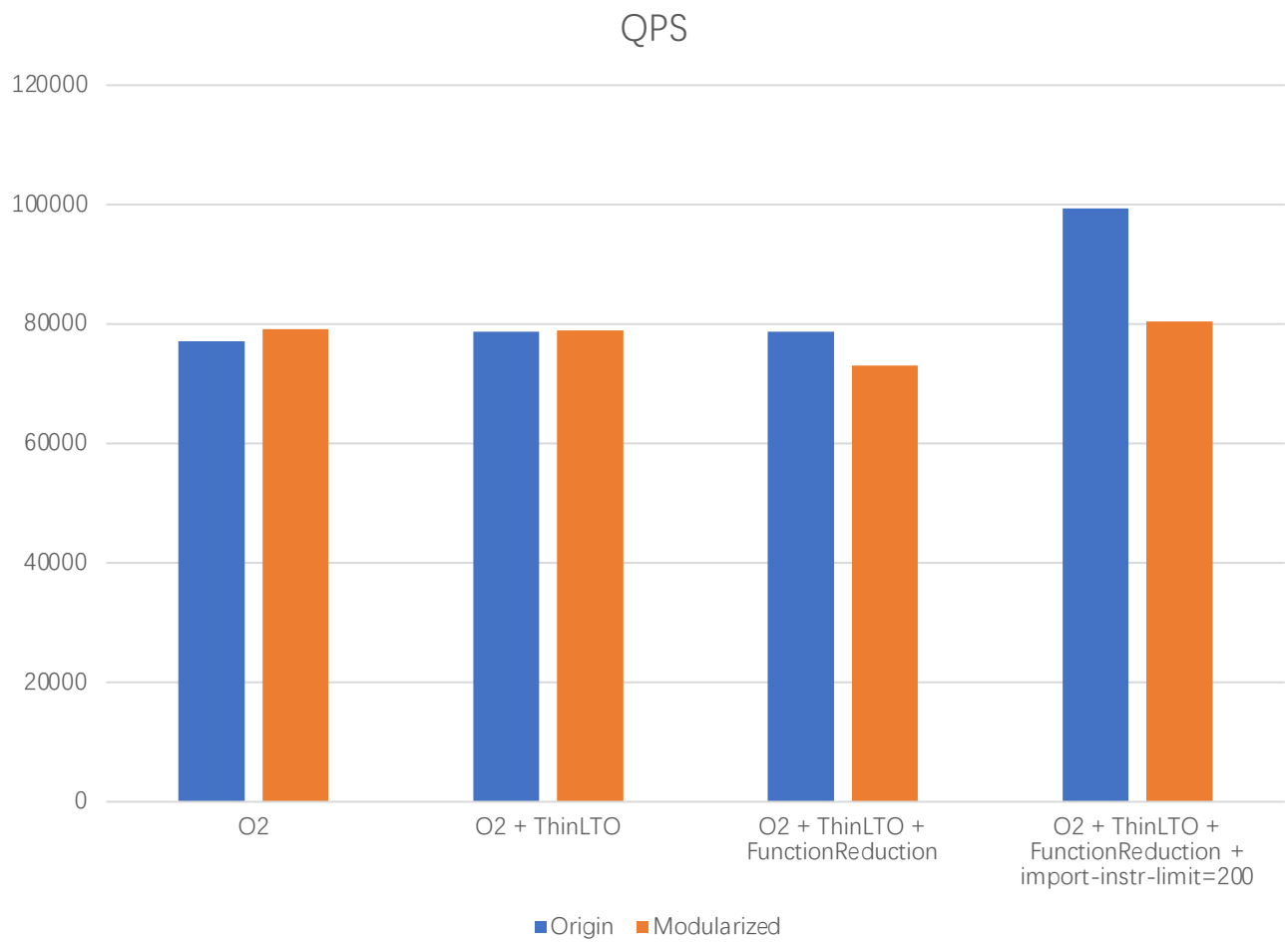
We measure the performance by the QPS of a async_simple based http server.



Case Study - Possible Improvement

Compiler Improvement – Function Reduction

Note that the modularized one has higher QPS in O2. We've measured many times and still get the same result. We suspect the reason may be the changed instruction layout, which causes different instruction cache behavior. But we didn't investigate it further and can't prove it. So the conclusion here is that the modules won't produce worse performance.



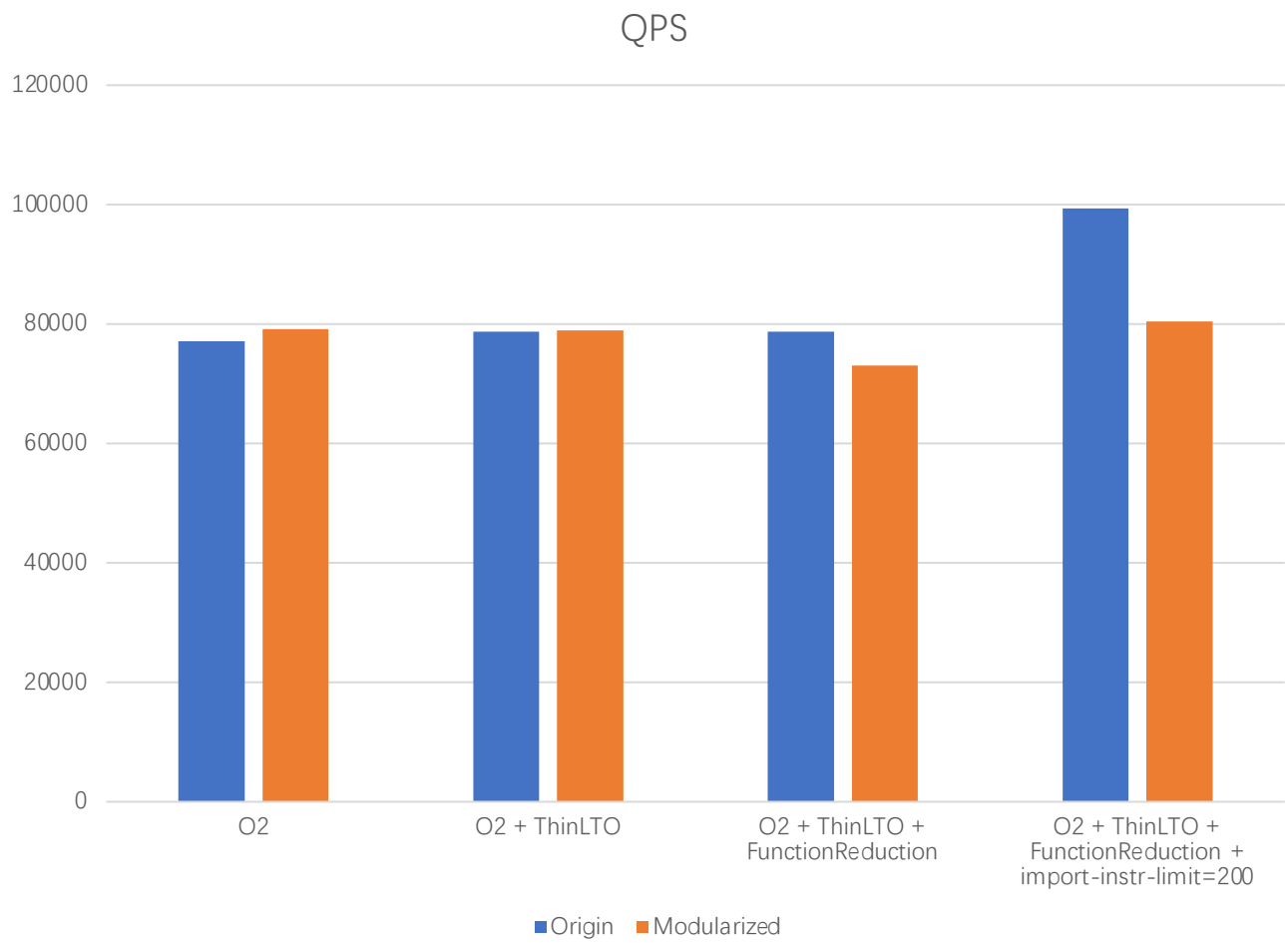
Case Study - Possible Improvement

Alibaba Cloud

Compiler Improvement – Function Reduction

The key point here is that:
with `-import-instr-limit=200`, the modularized one get higher QPS than the original ones without `-import-instr-limit`.

The conclusion here is:
We can tune the behavior of ThinLTO to mitigate the performance regression caused by function reduction.



Possible Improvement

Alibaba Cloud

Compiler Improvement – GMF Elision

The language spec allows the vendor to discard unreachable declarations in the global module fragment.

Possible Improvement

Alibaba Cloud

Compiler Improvement – GMF Elision

The language spec allows the vendor to discard unreachable declarations in the global module fragment.

This is not implemented yet. So we have potential improvement on code sizes and so the compilation speed.

Possible Improvement

Alibaba Cloud

Compiler Improvement – GMF Elision

The language spec allows the vendor to discard unreachable declarations in the global module fragment.

This is not implemented yet. So we have potential improvement on code sizes and so the compilation speed.

(From my draft experimental implementation, it looks like we can’t get significant improvement on compilation speed since the unused functions will not be imported to the middle end)

Possible Improvement

Alibaba Cloud

Library Improvement – reduce inline function

In headers, we would mark some functions as inline functions to avoid multiple definition error.

Possible Improvement

Alibaba Cloud

Library Improvement – reduce inline function

In headers, we would mark some functions as inline functions to avoid multiple definition error.

But this is not necessary in modules. A function defined in module units won't cause multiple function problem. So it is helpful for the compilation speed to remove inline function in modules.

Possible Improvement

Alibaba Cloud

Library Improvement – reduce inline function

In headers, we would mark some functions as inline functions to avoid multiple definition error.

But this is not necessary in modules. A function defined in module units won't cause multiple function problem. So it is helpful for the compilation speed to remove inline function in modules.

Note1: the compiler can't ignore inline specifier automatically due to ABI requirement.

Possible Improvement

Alibaba Cloud

Library Improvement – reduce inline function

In headers, we would mark some functions as inline functions to avoid multiple definition error.

But this is not necessary in modules. A function defined in module units won't cause multiple function problem. So it is helpful for the compilation speed to remove inline function in modules.

Note1: the compiler can't ignore inline specifier automatically due to ABI requirement.

Note2: the member function defined within a class is not implicitly inline if it is attached to a named module.

Possible Improvement

Alibaba Cloud

Library Improvement – instantiate hot templates

Given the previous introduction for the reason
why the speedup for templates are so satisfying.

We may wondering if we can cache something to
speedup it further.

However, it is not so useful and here is an simple experiment.

Possible Improvement

Alibaba Cloud

Library Improvement – instantiate hot templates

For this example, the module consumer get `0.1s` to compiler under O0.

```
import std.example;
int main() {
    std::vector<int> v;
    v.push_back(3);
    v.push_back(6);
    v.push_back(8);
    v.emplace_back(5);
    v.emplace_back(1);
    std::sort(v.begin(), v.end());
    std::printf("Is vector sorted? %d",
                std::is_sorted(v.begin(),
                               v.end()));
    return 0;
}
```

```
module;
#include <algorithm>
#include <cstdio>
#include <vector>
export module std.example;
export namespace std {
    using std::vector;
    using std::sort;
    using std::is_sorted;
    using std::printf;
}
export {
    using ::operator new;
}

#if defined(__GLIBCXX__) || defined(__GLIBCPP__)
export namespace __gnu_cxx {
    using __gnu_cxx::operator==;
    using __gnu_cxx::operator!=;
    using __gnu_cxx::operator-;
    using __gnu_cxx::operator<;
}
#endif
```

Possible Improvement

Alibaba Cloud

Library Improvement – instantiate hot templates

With the cached instantiations, the module consumer now takes `0.07s` to complete. We saves 30% for the time! Looks great, isn't it?

```
1   module;
2   #include <algorithm>
3   #include <cstdio>
4   #include <vector>
5   export module std.example;
6   export namespace std {…
12  export { using ::operator new; }
13
14  #if defined(__GLIBCXX__) || defined(__GLIBCPP__)
15  export namespace __gnu_cxx {…
21  #endif
22
23  int __stl_cache() {
24      std::vector<int> v;
25      v.push_back(1);
26      v.emplace_back(1);
27      std::sort(v.begin(), v.end());
28      std::printf("Is vector sorted? %d",
29                  std::is_sorted(v.begin(),
30                                 v.end()));
31  }
```

We can instantiate the templates inside the module.

Possible Improvement

Alibaba Cloud

Library Improvement – instantiate hot templates

However, it is not scalable:

- We can't predict the usage precisely.
- If we instantiate a lot of usages, we need a lot of codes. It is hard to maintain.
- Even if we reduce the codes we need to write by macro magics, the code sizes will be exploded in the end. For the example, the size change of the object file for the module unit is (4k -> 52k). Yes, it is 13x larger than before.
- The speedup will be reduced within optimization.

```
1   module;
2   #include <algorithm>
3   #include <cstdio>
4   #include <vector>
5   export module std.example;
6   export namespace std {…
12  export { using ::operator new; }
13
14  #if defined(__GLIBCXX__) || defined(__GLIBCPP__)
15  export namespace __gnu_cxx {…
21  #endif
22
23  int __stl_cache() {
24      std::vector<int> v;
25      v.push_back(1);
26      v.emplace_back(1);
27      std::sort(v.begin(), v.end());
28      std::printf("Is vector sorted? %d",
29                  std::is_sorted(v.begin(),
30                                 v.end()));
31  }
```

We can instantiate the templates inside the module.

Possible Improvement

Alibaba Cloud

Library Improvement – instantiate hot templates

So, just don't use this trick.
It is a trap.

```
1   module;
2   #include <algorithm>
3   #include <cstdio>
4   #include <vector>
5   export module std.example;
6   export namespace std {…
12  export { using ::operator new; }
13
14  #if defined(__GLIBCXX__) || defined(__GLIBCPP__)
15  export namespace __gnu_cxx {…
21  #endif
22
23  int __stl_cache() {
24      std::vector<int> v;
25      v.push_back(1);
26      v.emplace_back(1);
27      std::sort(v.begin(), v.end());
28      std::printf("Is vector sorted? %d",
29                  std::is_sorted(v.begin(),
30                                 v.end()));
31  }
```

We can instantiate the templates inside the module.

Possible Improvement

Alibaba Cloud

The improvement in the long term – embedded IR in BMI

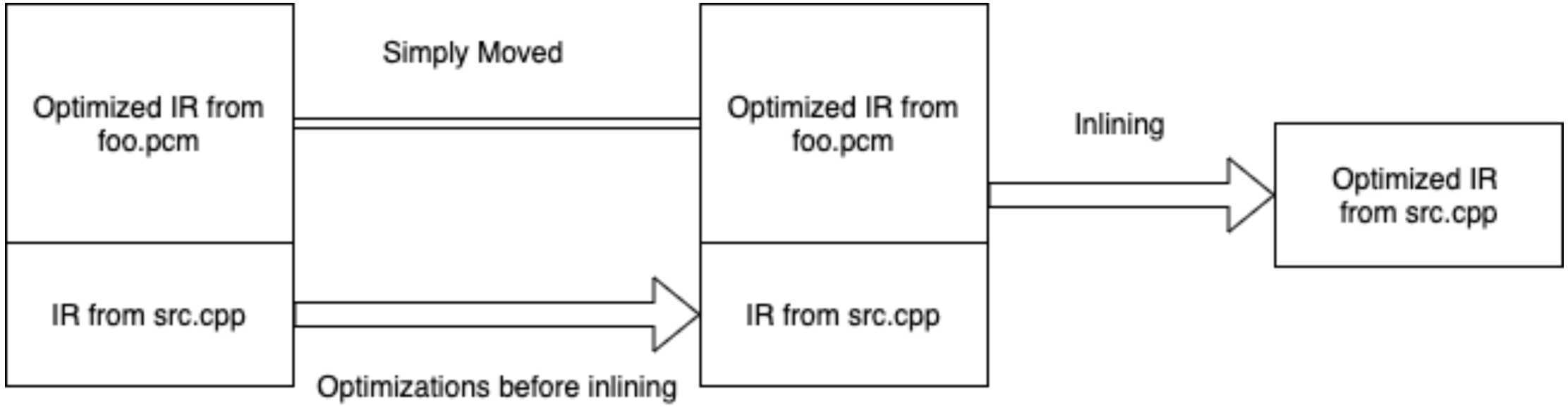
It looks a little bit frustrating that we'll get less speedup within optimizatations. Can we do anything to improve it without hurting performances?

It is possible. As long as we can embed the optimized IR in the BMI, so when we generate IR in the frontend, we can import the optimized IR directly instead of generate it from scratch. Then the optimized IR will be processed by inliner and all the other optimization will be skipped. In this way, we should get a much better compilation speedup within optimization.

Possible Improvement

Alibaba Cloud

The improvement in the long term – embedded IR in BMI



Given optimizations are expensive, we should get more speedup in this manner.

But this may not be able to implemented in short time.

Alibaba Cloud

Thanks for watching!