UNIVERSITY OF CALIFORNIA,
IRVINE

Modularization of C++ Applications Based on C++ 20 Modules

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Canyang Shi

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to Professor Joshua Garcia, Professor Crista Lopes, Professor Sam Malek, and all the professors and friends that helped me during my master's years.

ABSTRACT OF THE THESIS

Modularization of C++ Applications Based on C++ 20 Modules

By

Canyang Shi

Master of Science in Software Engineering

University of California, Irvine, 2022

Assistant Professor Joshua Garcia, Chair

As one of the most popular programming languages, C++ is characterized by its unique header-file mechanism that provides an effective way to access the interface of a library. However, this header-based mechanism also has its weaknesses. For instance, compilers have to perform redundant work which leads to poor compiling performance, developers should write their code carefully to avoid macro collisions, declarations and implementations are separated into multiple files which increase the complexity of the project, etc. To mitigate these challenges, the C++ standards committee proposed the modules feature of C++20 in 2020, which is introduced as an improvement of the traditional header file mechanism. C++20 modules provide a better way to encapsulate codes and address most deficiencies of header files. However, since the module feature is quite new, its advantages and potential challenges are not well-understood. On the other hand, most existing C++ applications are still based on header files and the include model, and there are not enough instructions on how to modularize a header-based app into a module-based app. To bridge these gaps, the paper discusses the influence of C++20 modules and proposes *H2M*, a new approach for conversion of a header-based C++ app to a module-based C++ app with better compiling performance. *H2M* starts by determining candidate source files for modularization. Next, it bundles up similar candidate source files and identifies appropriate dependencies. Finally, *H2M* generates the corresponding module-based app of the given header-based app. Our empirical studies verify the effectiveness of C++ 20 modules in improving compiling performance and the feasibility of code migration. Besides, the empirical studies on several header-based C++ applications demonstrate the effectiveness of *H2M*.

Chapter 1

Introduction

1.1 Background

Most software is based upon a number of external or internal libraries. In C and C++, a library is accessed by a file with an *#include* directive that includes the appropriate header files. However, because this header-based mechanism needs the compiler to preprocess the content of the header files transitively, it shows some significant weaknesses when getting access to the interface of a library. These weaknesses include low compile-time scalability, fragility, ambiguous interfaces, etc. To address these issues, an improvement or replacement for header files is an extreme necessity.

In 2020, the C++ standards committee proposed the Draft International Standards (DIS) of C++ 20 [10], one of the biggest updates of this thirty-seven-year-old programming language. A lot of new features were introduced in this new version, such as coroutines, ranges, modules, constraints and concepts. Among these new features, the modules feature stands out because it is designed as an improvement and eventual substitution of the traditional header-based model. In the C++ 20 module system, each file can be exported as a module with an *export* directive and then accessed by other files with an *import* directive. Since each module is compiled as a separate, standalone entity, this semantic import model overcomes many of the defects inherent to using traditional *#include* directives to access a library. According to the official documents of GCC and Clang [1] [5] [6], modules (1) reduce compile time significantly, (2) provide proper isolations between interfaces and implementations, (3) provide a new way to encapsulate codes that makes programs more comprehensible.

Nevertheless, since C++ 20 is a quite new version, the module system has not been discussed and analyzed in depth. Some questions may pop up in people's minds. What are the similarities and differences between C++ 20 modules and other module systems such as the Java Platform Module System (JPMS), a similar module system in Java? What benefits can C++ 20 modules bring to software and software developers? What potential challenges we may encounter when using C++ 20 modules? To answer these questions, this paper (1) gives a brief introduction of JPMS and compares it with C++ 20 modules, and (2) discusses the benefits and challenges of C++ 20 modules. On the other hand, converting a traditional header-based C++ application to a well-structured, module-based application is a tedious and error-prone work, while there is no existing approach or tool assisting developers with modularization. To bridge the gap, I have proposed *H2M*, an approach aiming to support developers to modularize header-based C++ apps.

More precisely, for a C++ header-based app, *H2M* (1) helps developers to determine candidate source files for modularization, (2) checks similarities among candidate source files and bundles up similar files, (3) adds appropriate dependencies. Finally, *H2M* generates the corresponding module-based C++ app. I have conducted an empirical study evaluating the effectiveness of this approach, including the ability of *H2M* in reducing compile time and increasing compile-time scalability.

1.2 Organization of the Thesis

The remainder of the paper is organized as follows. Chapter 2 introduces the similar module system in Java, the Java Platform Module System (JPMS), and compares it with C++ 20 modules. Chapter 3 analyzes some shortcomings of the existing header-based model in comparison to the benefits of C++ 20 modules, and then presents the usage of modules. Chapter 4 describes the modularizing approach in detail and explains the reason why it can reduce compile time and improve scalability. Chapter 5 shows the empirical evaluations and results. Chapter 6 presents further discussions about our findings. Chapter 7 presents some related work. Chapter 8 concludes the paper.

Chapter 2

Java Platform Module System

2.1 Overview of Java Platform Module System

Some other programming languages have already supported modularization since several years ago. For instance, a similar module system, Java Platform Module System (JPMS), was introduced in Java 9 [20]. JPMS aims to support architecture-based software development [2] [17]. Specifically, it enables Java developers to design a series of architectural constructs including modules, module directives and interfaces, which (1) provide better encapsulation of software, (2) improve maintainability of software, and (3) improve security by reducing attack surfaces.

In JPMS, Java applications can be organized in modules. A module is a higher level of abstraction containing a uniquely named group of packages with some other resources [15]. Each application has an extra configuration file called *module-info.java* which contains module declarations specifying module dependencies and exposed resources. Specifically, those relationships are depicted by a set of directives: *requires*, *exports*, *opens*, *provides*, *uses*. The *requires* directive specifies dependencies on other modules. Both *exports* and *opens* specify

```
module alice{
    requires service;
    exports com.example.alice.utils;
    opens com.example.alice.network;
    uses com.example.service.Srv;
}

module bob{
    requires service;
    provides com.example.service.Srv with com.example.bob.impl.ImplService;
}

module service{
    exports com.example.service;
}
```

Figure 2.1: An example of *module-info.java* with three modules.

the exposed packages. The difference is that an exported package is accessible at both runtime and compile time, while an opened package is only accessible at runtime. The *provides* directive specifies that a certain interface or abstract class is provided by this module as a service, and *uses* specifies the service object of an interface or abstract class used by the module.

Figure 2.1 shows some examples of module declarations in *module-info.java.* Three modules, *alice*, *bob* and *service*, are declared. Module *alice* has a dependency on module *service.* It exports the package *utils* and opens another package *network.* Besides, module *alice* uses the service object of class *Srv* provided by module *bob.* Module *bob* also has a dependency on module *service.* It provides *Srv* as a service with an implementation *ImplService.* Finally, module *service* exports the abstract class *Srv* to the public.

Table 2.1: Comparison between C++ 20 modules and JPMS.

System	**C++20 Module System**	**JPMS**
Similarities	Provides better encapsulation. Improves maintainability.	Provides better encapsulation. Improves maintainability.
Differences	Provides a replacement for header files. Does not involve any extra file. Supports one type of module dependency. Reduces the redundant work performed by the preprocessor.	Provides a higher level of abstraction. Involves a configuration file. Supports five types of module dependencies. Reduces the size of codes loaded at runtime.

2.2 Comparison between C++ 20 Modules and JPMS

There are some similarities between JPMS and the module system of C++ 20. For example, both of them can provide better encapsulation and improve software maintainability to a large extent. However, the main goals of these two module systems are different. JPMS is proposed in order to provide a better way to encapsulate codes in terms of architecturally significant elements, while C++ 20 modules are proposed as an improvement and eventual replacement for traditional header files. Due to the different goals, these two systems differ in various aspects. Table 2.1 lists the similarities as well as the differences between JPMS and the module system of C++ 20.

(1) In JPMS, a module is not a specific file but a higher level of abstraction containing a group of packages. Each module describes a software component. In comparison, a C++ 20 module is a standalone entity compiled from a single file. Each module serves as an interface of a library which can be accessed by other files.

(2) JPMS involves an extra configuration file, *module-info.java*, which specifies module dependencies and exposed resources. As a system evolves, architectural inconsistencies may occur since the as-conceived architecture specified by *module-info.java* may not match the

as-implemented architecture of the software [7]. Therefore, architectural recovery plays an important role in preventing architectural decay [3] [11] [16]. For the C++ 20 module system, on the other hand, there is no extra file needed. Module declarations and dependencies are included at the beginning of a file.

(3) JPMS supports five types of module dependencies, while in the module system of C++ 20, only the *import* dependency is supported. Because of the limited type of module dependency supported in C++ 20, developers may meet challenges during the code migration process of complex C++ projects with complicated file dependencies.

(4) JPMS can mitigate software bloat by reducing the size of codes loaded at runtime. For example, Java 9 itself is modularized, therefore, software developers can only require insignificant part of the Java Runtime Environment system modules. For C++ 20 modules, compile time is reduced due to the reduction of the amount of work performed by the preprocessor.

Chapter 3

C++ 20 Modules

3.1 Shortcomings of Traditional Header Files

Before modules were introduced in C++ 20, the C family of languages did not support modularization. The traditional way of getting access to a certain library is by using the *#include* directives, i.e., including the corresponding header files at the very beginning of the program. However, the header file mechanism provides an extremely poor way to access the API of a library due to the following reasons.

(1) **Poor Compiling Performance**: When a header is included, the compiler not only preprocesses and parses the text in that header, but also deals with every header it includes in the same way recursively. This process is repeated for every translation unit of the application, which involves a huge amount of redundant work. Figure 3.1 shows a typical application that makes the compiler perform redundant work. Specifically, the main program contains three header files, all of which include three shared header files. When building the application, the compiler must deal with each *#include* directive, so a great deal of time is wasted on unnecessary, repetitive work. Generally speaking, for a project with N translation

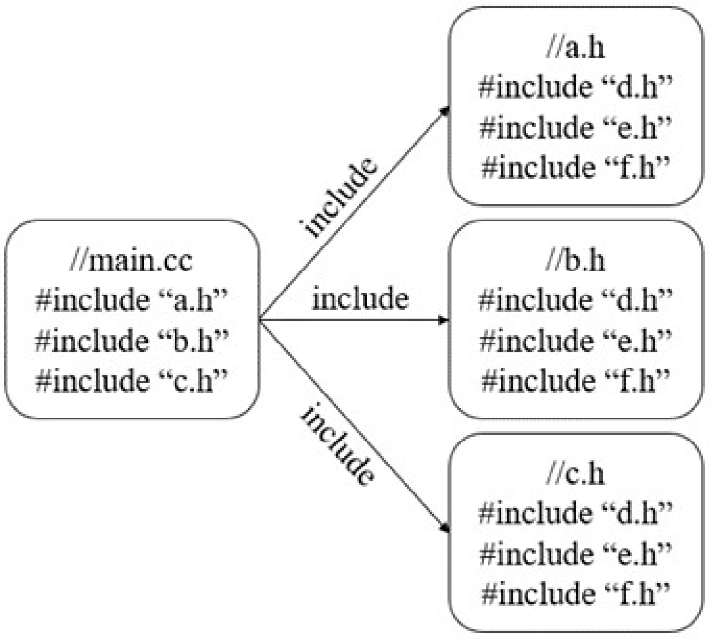


Figure 3.1: A typical application making the compiler perform redundant work.

units, each of which has M header files, the workload of the compiler is $M \times N$ even if most headers are shared by different translation units.

(2) **Macro Collisions**: Each *#include* directive is considered as a textual inclusion by the preprocessor, so it is subject to active macro definitions. When an active macro definition collides with another definition in the library by coincidence, it may lead to the failure of compilation or even severer problems. A more common case is that the headers of multiple libraries interact due to macro collisions. In that case, developers have to add extra *#undef* directives or reorder the existing *#include* directives to avoid collisions.

(3) **Ambiguous Libraries**: In a C-based language, the boundaries of software libraries are ambiguous, bringing challenges to developers who want to build software tools that work well with libraries. For example, it is hard to match a particular library with the header files that belong to it. In addition, it is hard to identify the declarations in headers that are actually designed as a part of the API rather than just a necessary part of the header file.

(4) **Code Duplication**: In the inclusion model, developers need to provide both the header

files (*xxx.h*) and their implementations (*xxx.cpp*). The former contains the declarations and the latter contains the corresponding definitions. As a result, there are lots of unnecessary repeats and redundancies which increase the complexity of the application.

3.2 Benefits of C++ 20 Modules

Modules provide an improved way to access the interface of software libraries by replacing the traditional inclusion model with a more efficient, more robust, more compact semantic model. Unlike a header file, a module is compiled as a separate and independent entity. Specifically, in the module system of C++ 20, the compiler loads a binary representation of the module, which is called a binary module interface (BMI), and then makes it accessible by the application directly. In this way, many defects of the inclusion model are eliminated.

(1) **Compiling Performance**: Compiling performance is improved due to the fact that each module is only compiled once. Importing an existing module to a translation unit is a constant-time operation which does not involve recursive procedures. As a result, the workload of the compiler is reduced from $M \times N$ to $M + N$.

(2) **Macro Collisions**: Each module is compiled as a separate entity that cannot affect the compilation of other modules. Thus, macro collisions are fundamentally avoided.

(3) **Ambiguous Libraries**: Modules serve as descriptions of the interface of software libraries, so software tools can just provide a module as a representation of the interface. On the other hand, because each module has its consistent preprocessor environment and can be compiled separately, software tools can ensure that they get the complete API of the library by importing the corresponding module.

(4) **Code Duplication**: Since declarations and definitions are both written in a single file,

redundant codes are removed. In this way, applications are simplified and the comprehensibility is largely increased.

3.3 Usage of C++ 20 Modules

The C++ 20 module system only has two kinds of directives so far, i.e., *export* and *import*. The *export* directive is used for a module declaration that specifies the name as well as the exposed contents. On the other hand, modules can be imported to other files with the *import* directive.

Figure 3.2 shows an example of using modules in C++ 20. A module is exported by adding the directive *export module* followed by its name (line 4), and the contents of a module are exported by adding the *export* directive before declarations (line 5). On the other hand, other files can import a module with the *import* directive followed by the name (line 10). For GCC, the example can be compiled with the command "g++ -std=c++20 -fmodules-ts hello.cc main.cc".

Traditional header files can still be accessed in a module through the global module fragment, which is a prefix of the module unit (line 2 and line 3 in Figure 3.2). Global module fragment enables developers to include header files when it is not possible to import them, especially when a header file uses preprocessing macros as a form of configuration.

Besides, a traditional header file can be imported into a module unit as a header unit (line 2 and line 3 in Figure 3.3). A header unit is, like a module, built in advance separately with a special flag (for example, "-fmodule-header" in GCC). It also produces a BMI and can be imported into both module and non-module files with the *import* directive.

```cpp
//hello.cc
module;
#include <iostream>
export module hello;
export void say_hello(){
    std::cout << "Hello!" << std::endl;
}

//main.cc
import hello;
int main(){
    say_hello();
    return 0;
}
```

Figure 3.2: An example of using modules in C++ 20.

```cpp
//main.cc
import <iostream>;
import "otherheader.h";
int main(){
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Figure 3.3: An example of importing header files in C++ 20.

Chapter 4

Approach for Modularization

This chapter presents further details of the modularizing approach, *H2M*, and explains the reason why it can reduce compile time and improve compile-time scalability to a large extent. The goal of *H2M* is to assist software developers to modularize a header-based application to a module-based application with better compiling performance and better encapsulation. As depicted in Figure 4.1, for a given header-based application as the input, *H2M* generates the corresponding module-based application as the output. Specifically, *H2M* consists of four steps including candidate determination, file bundling, dependency identification, and code migration. The rest of this chapter describes each step in depth.

4.1 Step 1: Candidate Determination

It is not always feasible or necessary to modularize all the source files of a header-based app. For example, for large and complex C++ apps, full modularization may be impossible because it is either too expensive or time-consuming, while modularizing only a part of the project can still bring enough benefits such as a significant reduction in compile time.

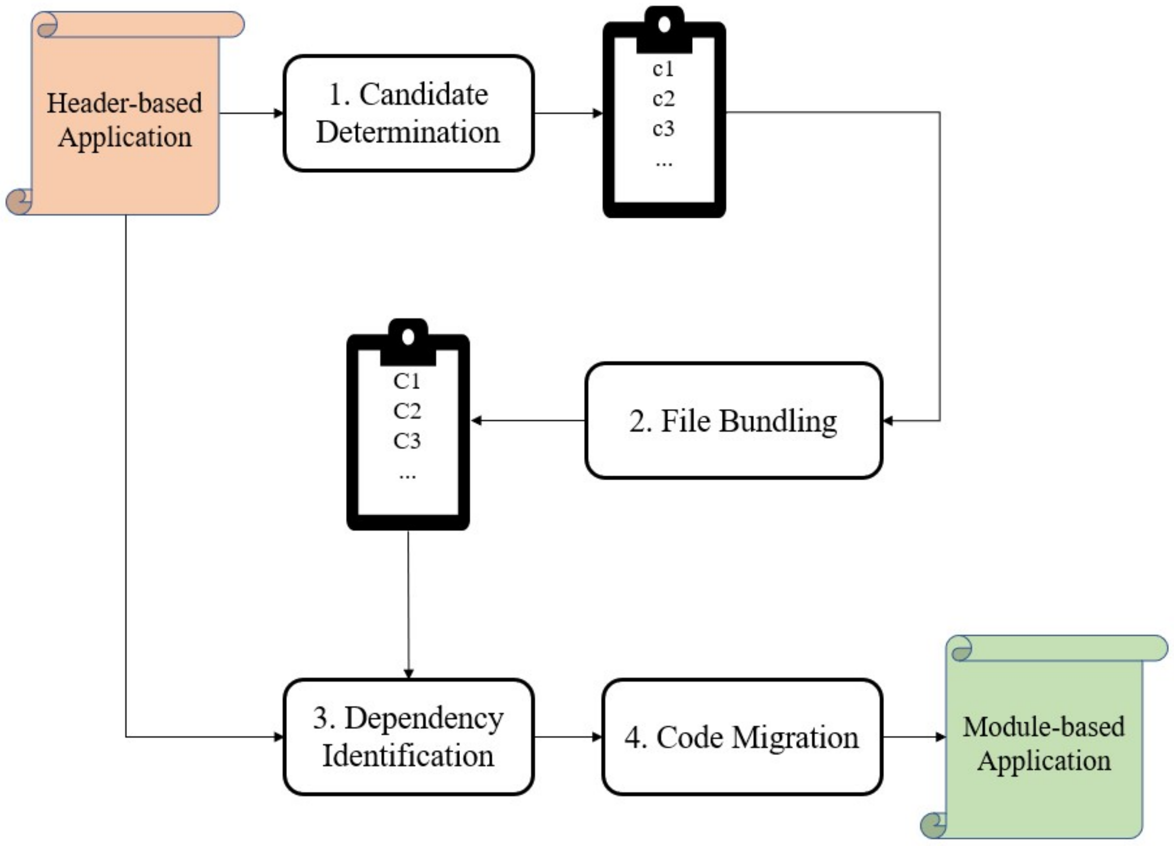


Figure 4.1: Overview of H2M.

Therefore, the first step to build a module-based app is determining the candidate source files that need to be modularized. Specifically, *H2M* determines a subset of source files based on a given threshold t_d that limits its size. These candidate files are the output and they will be passed to the second step for further modularization. The routine of candidate determination is presented in Algorithm 1.

Algorithm 1: Candidate Determination

```
input  : a header-based app and a threshold
output: a set of candidate source files
1 let $s$ denote the set of files of the header-based app and let $t_d$ denote the threshold;
2 while less than $|s| \times t_d$ files are marked do
3     find the file $f$ in $s$ which is dependent by most files;
4     mark $f$ as a candidate source file;
5     remove $f$ from $s$;
6 end
```

In each iteration, *H2M* looks for the file with most dependencies and marks it as a candidate source file that needs to be modularized. Transitive dependencies are also taken into account. For example, if file a depends on file b, and file b depends on file c, then file c is considered dependent by both a and b. The threshold t_d describes the percentage of files which developers hope to modularize, and the selection of t_d is based on experience. Larger thresholds result in more thorough modularization. In particular, the application is fully-modularized for $t_d = 100\%$.

Figure 4.2 provides an illustrative example of the candidate determination process with the threshold $t_d = 60\%$. Figure 4.2(a) shows the structure of the header-based app. Each node represents a single file and each arrow represents a file dependency, i.e., *include* dependency. Because the app consists of eight files and the threshold is 60%, the algorithm will terminate after five files are marked. At the very beginning, we have $s = \{a, b, c, d, e, f, g, h\}$. In the first

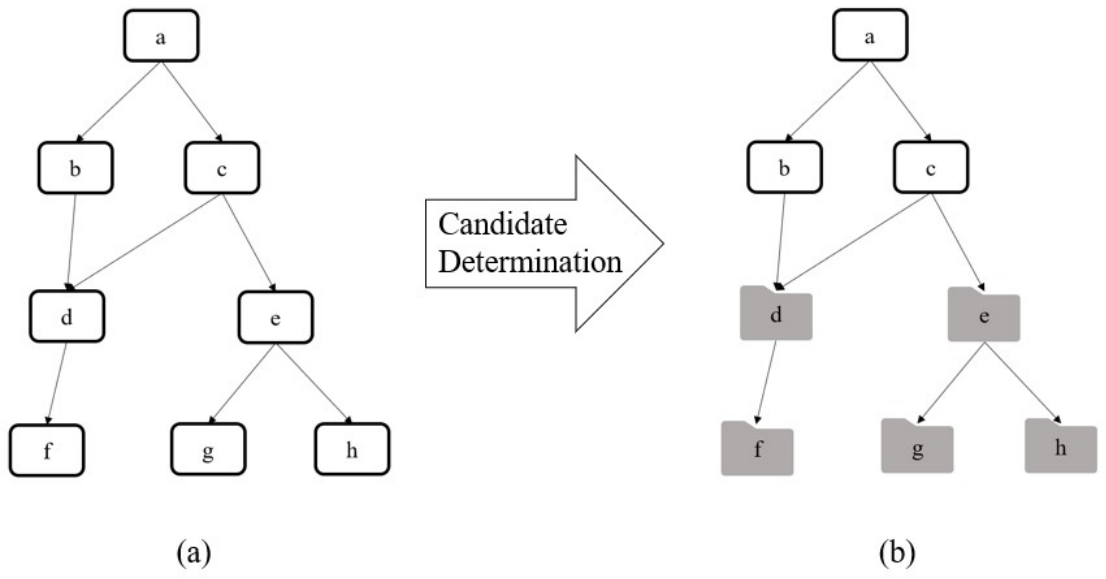


Figure 4.2: An illustrative example of candidate determination.

iteration, file f is marked as a candidate source file because it is used by most files $\{a, b, c, d\}$, including one direct dependency from file d and three transitive dependencies from file a, file b and file c. Then, after removing f from set s, we have $s = \{a, b, c, d, e, g, h\}$. Similarly, in the next four iterations, file d, g, h, and e are marked as candidates successively. In the end, we obtain a subset of files, $\{d, e, f, g, h\}$, which is determined as the set of candidate source files. These files will be passed to the file bundling step for further modularization.

4.2 Step 2: File Bundling

In this step, *H2M* determines module contents by clustering the candidate source files. To achieve this, *H2M* checks the dependency-similarities among the candidate files determined in Step 1, and then bundles up source files with similar file dependencies. These bundled files are passed to the following step for further modularization.

4.2.1 Dependency-similarity

In order to quantify the dependency-similarity between two files, I introduce the Jaccard similarity coefficient. Specifically, each file can be represented by a set containing itself as well as all the files that depends on it (including transitive dependencies). For instance, if file a depends on file b, and file b depends on file c, then file c can be represented by the set $\{a, b, c\}$. In this way, the similarity between two files can be quantified by calculating the Jaccard similarity coefficient:

$$similarity(f_1, f_2) = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}$$

For example, if file f_1 is represented by the set $\{a, b, c, d, e\}$ and file f_2 is represented by the set $\{a, c, e, f, g\}$, then the dependency-similarity between f_1 and f_2 is

$$similarity(f_1, f_2) = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|} = \frac{|\{a,c,e\}|}{|\{a,b,c,d,e,f,g\}|} = \frac{3}{7}$$

A threshold t_b should be provided as a standard determining whether two files are similar enough. If the similarity between two files is greater than t_b, then they are considered as similar files and are bundled together.

4.2.2 Algorithm of File Bundling

Based on the aforementioned dependency-similarity quantification, the routine of file bundling is presented in Algorithm 2.

Algorithm 2: File Bundling

```
input  : a set of candidate source files and a threshold
output: a set of merged candidate source files
1  let $s$ denote the set of candidate source files and let $t_b$ denote the threshold;
2  while $s$ is not empty do
3      select a file $f$ from $s$, check if the dependency-similarity between file $f$ and
         another file in $s$ is greater than $t_b$;
4      if a qualified file $g$ is found then
5          bundle up $f$ and $g$ into a new file $fg$;
6          remove $f$ and $g$ from $s$;
7          add $fg$ into $s$;
8      else
9          remove $f$ from $s$;
10     end
11 end
```

In each iteration, *H2M* checks if two candidate source files are similar enough, i.e., the dependency-similarity is greater than the threshold t_b. The threshold reflects how similar two files should be if they need to be merged, and the selection of it is also based on experience. Therefore, a smaller threshold results in fewer but larger modules, while a larger threshold results in more but smaller modules. Then, for similar candidate files, *H2M* bundles them up. To achieve this, *H2M* obtains the set representations of the two files, and replaces them with their union.

Figure 4.3 shows an illustrative example of the file bundling process with the threshold $t_b = 50\%$. Figure 4.3(a) presents the output of candidate determination. There are five candidate source files at first, i.e., $s = \{d, e, f, g, h\}$. In the first iteration, for example, file d is selected, and *H2M* checks the dependency similarities between d and each of the other

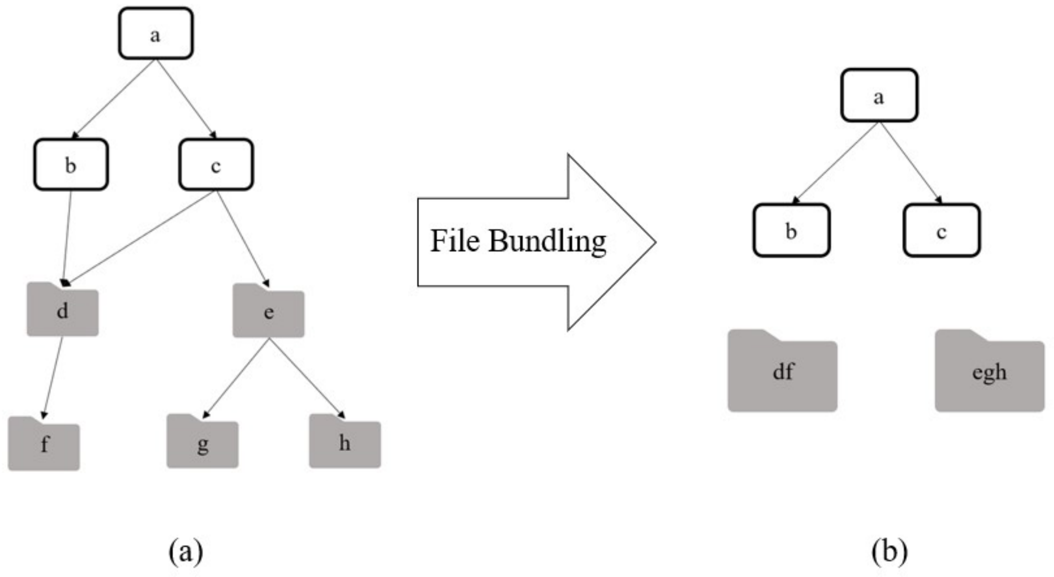


Figure 4.3: An illustrative example of file bundling.

candidate file. Since d and e are not similar enough, i.e., $similarity(d, e) = 40\% < t_b$, they should not be bundled together. However, we have $similarity(d, f) = 80\% > t_b$, so d and f are merged into a larger file represented by df. At this time, we have $s = \{df, e, g, h\}$. Similarly, g and h are bundled up as gh in the second iteration, then e and gh are bundled up as egh in the third iteration (as shown in Figure 4.3(b)). At this time, $s = \{df, egh\}$ and $similarity(df, egh) = 40\% < t_b$. As a result, there is no bundling operation and file df is removed from s in the fourth iteration. Finally, the only element egh is also removed in the fifth iteration and the algorithm terminates. In the end, *H2M* bundles up the five candidate source files into two files, df and egh.

4.3 Step 3: Dependency Identification

After obtaining the set of bundled candidates, *H2M* identifies file dependencies related to them. For each file, *H2M* checks all the headers it includes according to the header-based app. If the file includes a candidate source file, then *H2M* replaces the include dependency with an import dependency upon the corresponding bundled candidate. For instance, in Figure 4.3(b), file *a* includes file *b* and file *c*, both of which are not candidates, therefore, no file dependency of a needs to be modified. For file *b*, *H2M* identifies its import dependency upon file *df* because it includes file *d* in the header-based app. Also, *H2M* identifies the dependencies of file *c* upon *df* and *egh*. Finally, both module contents and dependencies have already been determined after this step.

4.4 Step 4: Code Migration

Based on the bundled candidates determined in Step 2 and the file dependencies identified in Step 3, the module-based app can be generated. Specifically, each bundled candidate file is exported as a module by adding the corresponding module declaration statements in it. Header-only files are exported as header units. On the other hand, *import* directives are added to files that depend upon modules. Finally, the given header-based app is converted to a module-based app (Figure 4.4).

4.5 Analysis of the Module-based App

The module-based app generated by *H2M* not only takes less time to build, but also shows better compile-time scalability. The reason is that for each header included in the header-based app, the C++ compiler needs to preprocess and parse the text in that header as well as all the headers it includes. For the header-based app, a lot of redundant work is performed by the compiler which takes a large amount of time, especially when many large headers are included in lower-layer files. On the contrary, importing a module is almost free in a module-based app. As depicted in Figure 4.4, each bold arrow describes an expensive include

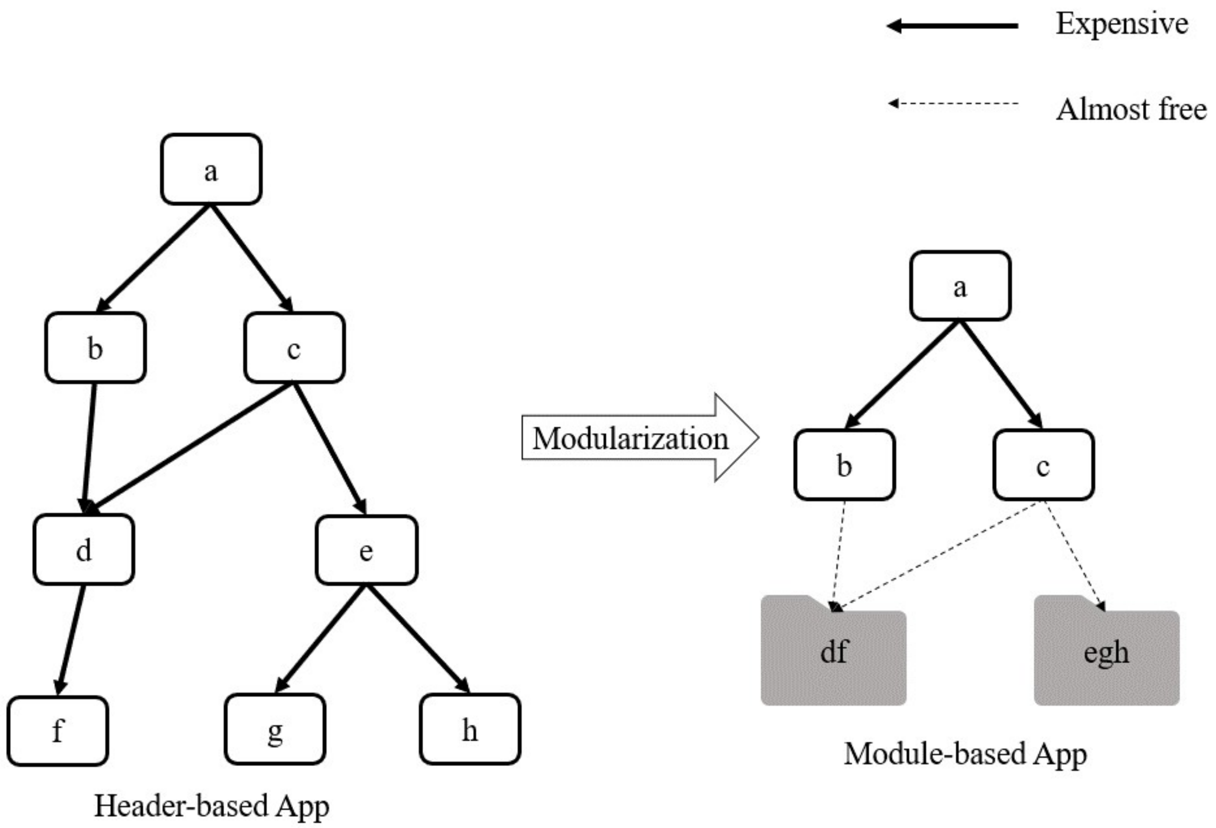


Figure 4.4: Overview of the modularization.

operation, and each dashed arrow describes a cheap import operation. After modularization, the number of bold arrows is reduced from eight to two, resulting in a significant decrease in build time.

On the other hand, for a header-based application, modification of a single file leads to recompilation of all the files that depend on it. For instance, if file h in Figure 4.4 is modified, then a set of files, $\{a, c, e, h\}$, needs to be rebuilt. In comparison, for the module-based application, only the corresponding module, egh, needs to be recompiled. Besides, as software evolves, more files may be added to the project. For each new file, it will not bring a significant increase to the total compile time if it can be built as a new module or merged into an existing module. Another case is that the new file itself is not built as a module, but it may depend on one or more existing modules, so compared with the header-based app, fewer expensive include operations are introduced to the module-based app. In both cases, the compile time of the module-based app grows much slower. As a result, the module-based app generated by *H2M* has better compile-time scalability.

Chapter 5

Empirical Studies

This chapter presents the results of empirical studies evaluating the module system of C++ 20 and the modularization approach *H2M*. For the C++ 20 module system, I have measured its influences on compile time, and the feasibility of code migration for simple projects. For *H2M*, I evaluated its effectiveness in converting header-based apps to module-based apps. I also evaluated how *H2M* can reduce compile time and improve compile-time scalability. Specifically, I focus on the following research questions.

- **RQ1:** How is compile time affected when header files are replaced by C++ 20 modules?
- **RQ2:** How feasible it is to migrate existing, simple C++ projects?
- **RQ3:** How effective is *H2M* in successfully converting header-based apps to module-based apps?
- **RQ4:** To what degree can *H2M*'s modularization reduce compile time and improve compile-time scalability?

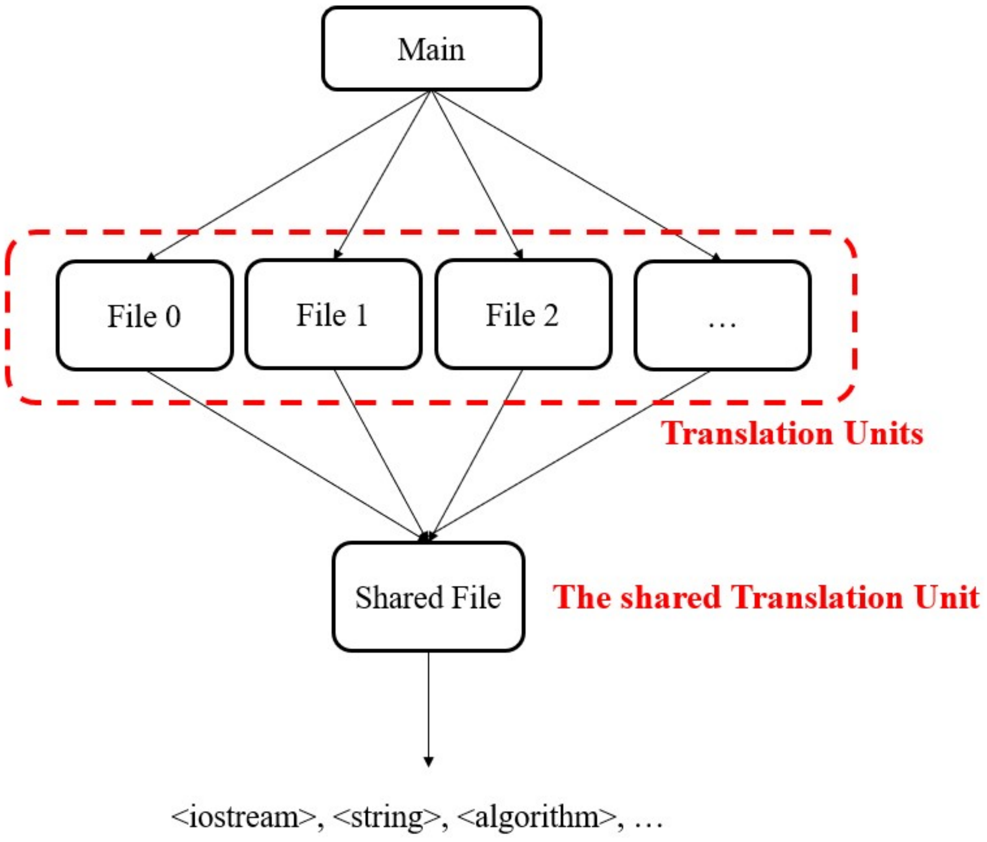

Main
File 0
File 1
File 2
…
Translation Units
Shared File
The shared Translation Unit
<iostream>, <string>, <algorithm>, …

Figure 5.1: The evaluating framework for the comparison of header files and modules.

5.1 C++ 20 Modules' Influences on Compile Time

To evaluate how compile time is affected when replacing header files with C++ 20 modules, I designed a demo project to compare the build time of a header-based program and a module-based program with respect to different numbers of translation units and shared header files. Figure 5.1 presents the structure of the demo project.

More precisely, the project contains a main file, a set of middle-layer translation units, and a shared translation unit. The arrows represent file dependencies, i.e., include dependencies in a header-based program and import dependencies in a module-based program. The main file depends on the middle-layer translation units, and the latter depends on the shared translation unit. Besides, a set of C++ standard headers are included in the shared file.

Table 5.1: Compile time of the header-based program and the module-based program

	Shared	**File 0**	**Main**	**Total**
header	0.26	0.26	0.25	**0.77**
module	0.31	0.01	0.01	**0.33**

# Header File × # Translation Unit = 1 × 1 (Seconds)

	Shared	**File 0**	**Main**	**Total**
header	0.43	0.42	0.43	**1.28**
module	0.47	0.02	0.02	**0.51**

# Header File × # Translation Unit = 6 × 1 (Seconds)

	Shared	**File 0**	**File 1**	**File 2**	**File 3**	**File 4**	**File 5**	**Main**	**Total**
header	0.26	0.26	0.26	0.25	0.25	0.26	0.26	0.25	**2.05**
module	0.29	0.01	0.02	0.02	0.02	0.02	0.01	0.01	**0.40**

# Header File × # Translation Unit = 1 × 6 (Seconds)

Based on this demo project, I compared the build time between a header-based program and a module-based program. The build time of each file can be obtained by adding an extra flag in the compiling command, for example, ”-ftime-report” in GCC. The results are presented in Table 5.1.

As the standard, the original project contains only one middle-layer translation unit (File 0), and one standard header is included in the shared file. In this case, the total compile time of the header-based program and the module-based program is 0.77s and 0.33s, respectively. Therefore, even for this pretty simple project, compilation takes less time when header files are replaced by C++ 20 modules.

When the number of standard header files included in the shared unit increases from one to six, the increase of compile time of the header-based program is 0.51s (1.28s in total), while the increase is only 0.18s for the module version (0.51s in total). The main cause of this difference is that the C++ compiler has to preprocess more header files when dealing with each source file (as described in Chapter 2). However, to build the module-based program, the compiler just spends little more time building the module based on the shared file, while importing the module in other files takes a constant amount of time. Therefore, the time of compiling File 0 and the main file remains almost unchanged.

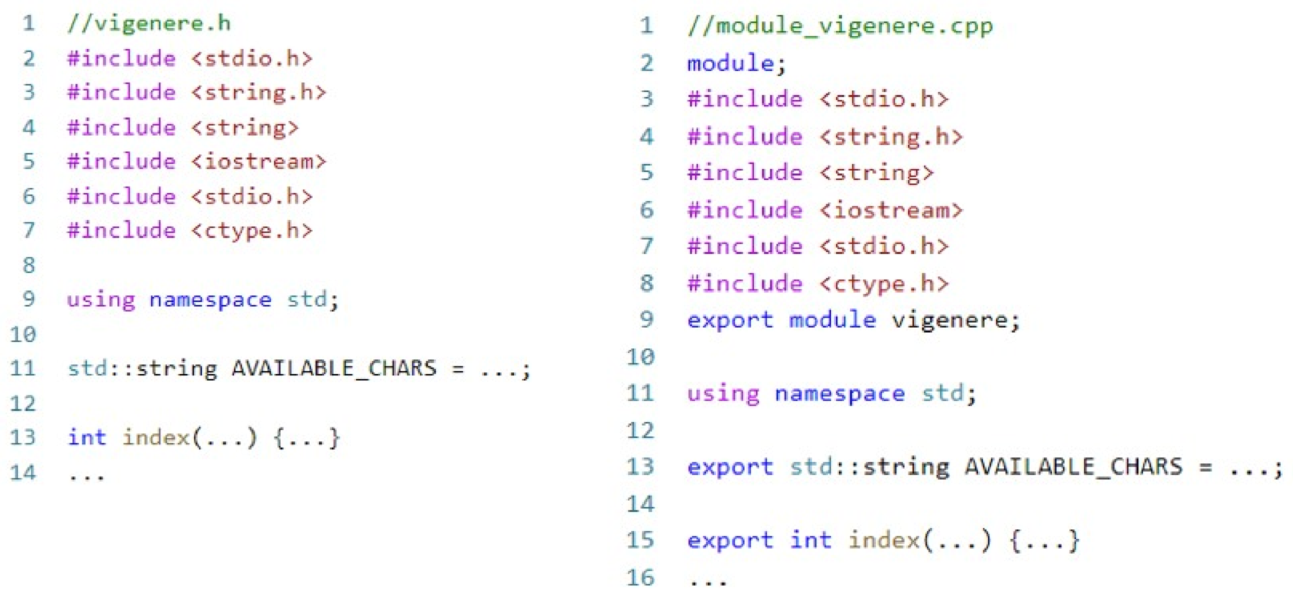
On the other hand, after increasing the number of middle-layer translation units from one to six, there is also a huger gap of compile time between these two programs. The total compile time of the header-based program and the module-based program is 2.05s and 0.40s, respectively. The reason is similar: the C++ compiler preprocesses and parses headers recursively but imports a module within constant time. Actually, for the header model, most time is spent on building translation units, while for the module model, most work is already finished after compiling the module.

In sum, for **RQ1**, the results of the empirical studies show that the module system of C++20 can reduce compile time to a large degree. The reduction of compile time is largely affected by the number of shared header files as well as the number of translation units.

5.2 Feasibility of Code Migration for Simple Projects

To evaluate the feasibility of code migration for real-world simple projects, I conducted a case study based on an open-source project on GitHub [18] and replaced its header files with modules by adding appropriate module directives. The project is an encryption framework with a pretty simple structure: the main file includes some C++ standard libraries together with a header file called *encrypt.h*, in which two other header files, *vigenere.h* and *b64.h*, are included. Both of the latter two header files only include C++ standard libraries.

In order for code migration, I exported *vigenere* and *encrypt* as two modules. In addition, module *vigenere* is imported in module *encrypt*, and the latter is imported in the main file. The *#include* directives of C++ standard libraries are preserved with the use of the Global Module Fragment. Figure 5.2 shows the code migration of *vigenere* as an example. Finally,



```
//vigenere.h
#include <stdio.h>
#include <string.h>
#include <string>
#include <iostream>
#include <stdio.h>
#include <ctype.h>

using namespace std;

std::string AVAILABLE_CHARS = ...;

int index(...) {...}
...
```

```
//module_vigenere.cpp
module;
#include <stdio.h>
#include <string.h>
#include <string>
#include <iostream>
#include <stdio.h>
#include <ctype.h>
export module vigenere;

using namespace std;

export std::string AVAILABLE_CHARS = ...;

export int index(...) {...}
...
```

Figure 5.2: The header-based file (left) and the module-based file (right).

the module-based project can be compiled and executed successfully, so for **RQ2**, the answer is that code migration is feasible for simple projects.

5.3 Effectiveness of H2M

This research question measures the effectiveness of *H2M* in modularizing a header-based app. To answer this question, I ran the approach on a real-world open-source project called TextBasedAdventure [22]. I modularized both the first commit and the latest commit, and checked if both module-based versions of these two commits can be compiled and executed successfully. As shown in Figure 5.3, TextBasedAdventure has two files, *Utilities* and *Globals*, which are used by almost all the other files in both commits. I set the thresholds $t_d = 88\%$ and $t_b = 50\%$. According to *H2M*, I bundled up these two files and exported them as a single module named *GlobalModule*. For both module-based versions, I can build and run them successfully. In sum, for **RQ3**, the empirical studies show that *H2M* is able to convert a header-based app to a module-based app successfully.

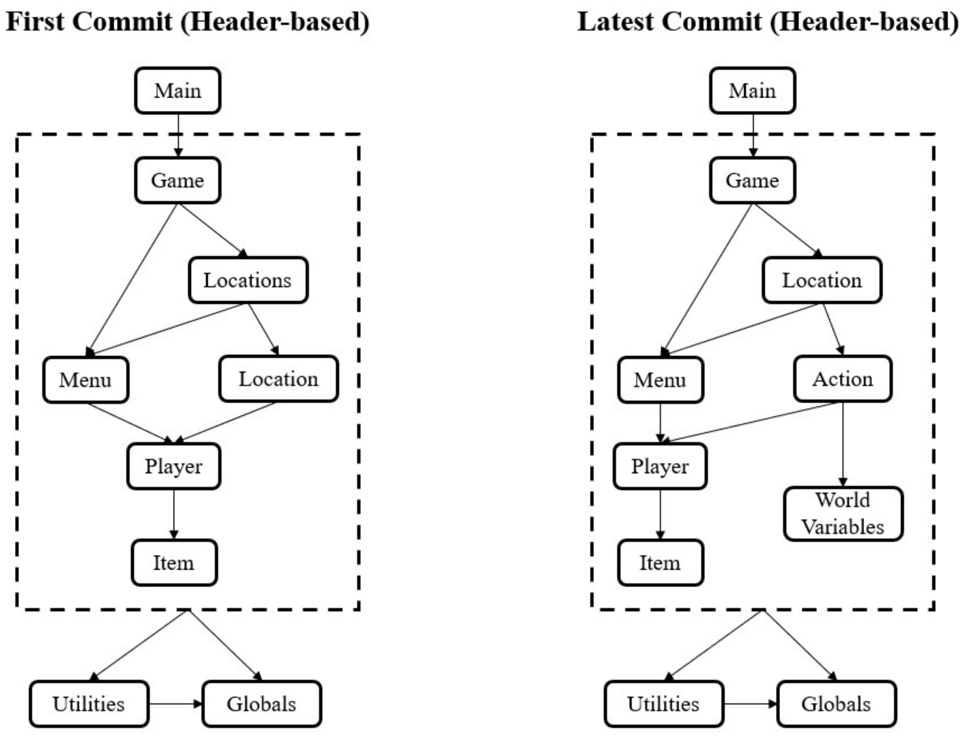


Figure 5.3: The structure of TextBasedAdventure.

5.4 H2M's Influences on Compiling Performance

To evaluate how *H2M*'s modularization can reduce compile time and improve scalability, I measured the compile time of the first commit and the latest commit of TextBasedAdventure [22], including both the header version and the module version (Table 5.2).

As shown in the table, the compile time of both versions increases more or less as the application evolves. More precisely, for the header-based version, the build time of the first commit is 3.09s, while it takes 4.18s to build the latest commit, and the increase of build time is 1.09s. As for the module-based project, the total compile time grows from 1.82s to 2.22s and the increase of build time is only 0.4s. In summary, after modularization, the total compile time is reduced by about 41%, and the increase of compile time is reduced by about 63%. Therefore, for **RQ4**, we can conclude that *H2M* can both reduce compile time and improve compile-time scalability to a large extent.

Table 5.2: Compile time before and after modularization

	Ut	**Pl**	**Me**	**Wo**	**Ac**	**Lo**	**Los**	**Ga**	**Ma**	**Total**
first	0.42	0.50	0.44	/	/	0.44	0.44	0.44	0.41	**3.09**
latest	0.48	0.56	0.49	0.47	0.50	0.57	/	0.59	0.52	**4.18**
difference	0.06	0.06	0.05	0.47	0.50	0.13	-0.44	0.15	0.11	**1.09**

Header-based Project (Seconds)

	M	**Ut**	**Pl**	**Me**	**Wo**	**Ac**	**Lo**	**Los**	**Ga**	**Ma**	**Total**
first	0.85	0.13	0.19	0.13	/	/	0.14	0.14	0.14	0.10	**1.82**
latest	0.94	0.14	0.22	0.15	0.11	0.15	0.18	/	0.21	0.12	**2.22**
difference	0.09	0.01	0.03	0.02	0.11	0.15	0.04	-0.14	0.07	0.02	**0.40**

Module-based Project (Seconds)

5.5 Threats to Validity

The main threat to the external validity of the evaluations is the projects I selected. All the experiments are based on relatively small projects, so *H2M* is not well-evaluated on large applications. I do not run the approach on sizable, real-world applications because the code migration process of such applications is challenging and time-consuming for people who do not maintain it. However, the conclusions based on small projects can be generalized. The reason is that the key idea of the modularization of *H2M* is reducing the amount of work performed by the compiler. For sizable, real-world applications, *H2M* can still determine the files that are mostly shared and export them into modules. Since the compiler does not need to preprocess shared header files multiple times any more, the compiling performance of large applications can also be improved.

Chapter 6

Discussion

As one of the new features of C++ 20, it is obvious that the module system brings a lot of benefits and possibilities such as better encapsulation, less compile time, better compile-time scalability, etc. In this paper, the influences of C++ 20 modules on compile time are evaluated and the feasibility of code migration is verified, and the results are both exciting.

However, on the other hand, C++ 20 modules bring us challenges as well. As the official document of Clang says [1], C++ modules are not designed to rewrite the world's codes, and it is not feasible to completely eliminate header files in the world. Actually, one big challenge is that the code migration process of complex projects may be difficult, especially for people who do not maintain them. Some existing projects can only be built in the context of a specific version of C++ and compatibility issues may occur if we simply replace some headers with modules. Besides, because the module system is a quite new feature of C++ 20, some existing tools or frameworks do not support it well.

Another challenge is that security issues may occur when C++ 20 modules are used improperly. Unlike some other existing module systems, this system does not involve any protection mechanism. For example, in the Android platform, permissions can be used to restrict access to a certain API. However, in the C++ 20 module system, the only thing developers can do is to decide whether a declaration should be exported. Therefore, when a module-based C++ application exports its internals excessively or imports external modules without verifying the authorities, malicious applications may access the private data, resulting in leakages of privacy or even some severer problems [12] [13] [14].

Despite the aforementioned engineering and security issues, the approach proposed in this paper can still benefit C++ developers to a large degree. Not only does it provide an effective way of modularization that reduces compile time and improves scalability significantly, but it also presents insights and instructions on the way to build new applications with modules. For now, C++ modules must interoperate with existing software libraries, but a gradual transition may occur in the future when *H2M* can assist software developers to build module-based applications.

Chapter 7

Related Work

A number of previous research work proposed different approaches or tools supporting software developers to determine software components from source codes. Tzerpos and Holt [21] proposed a clustering-based software recovery approach named *ACDC* that utilizes a system's structural characteristics to determine architectural components. Garcia et al. [4] proposed an architectural recovery tool, *ARC*, which groups entities based on system concerns. Gholam et al. [19] proposed a method to identify software components and their responsibilities based on the clustering of use cases.

A group of studies focuses on the detection and prevention of software inconsistencies. Some of them identify software inconsistencies by reverse engineering the descriptive architecture from the source code followed by comparing it with the prescriptive architecture. Ghorbani et al. [7] proposed *DARCY*, an approach that automatically detects and repairs inconsistent dependencies within Java applications based on static analysis. The work of Hammad et al. [8] [9] determines and enforces the least-privilege architecture in Android.

Unlike the aforementioned approaches and tools, *H2M* is the only approach for modularization of C++ header-based apps that focuses on improving compiling performance.

Chapter 8

Conclusion

There is no denying that the release of C++ 20 shows the vigor and popularity of this thirty-seven-year-old programming language. In particular, as one of the new features of C++ 20, the module system provides a new way to encapsulate codes and make header files unnecessary. This paper presents an overview of C++ 20 modules and proposes *H2M*, an approach of modularizing a header-based app. For C++ 20 modules, our empirical studies confirm their benefits of improving compiling performance, as well as the feasibility of code migration for simple projects. Besides, our experiment on a real-world project shows the effectiveness of *H2M* in successfully modularizing a header-based app to a module-based app with less compile time and better scalability. Specifically, *H2M* achieves a 41% reduction of compile time and a 63% reduction of compile time increment. Despite the fact that migrating complex projects may be hard and some tools do not support C++ modules well, it is true that the module system of C++ will become better and bring more convenience to developers in the short future.

Bibliography

[1] Clang. Official Documents. https://clang.llvm.org/docs/Modules.html, 2020.

[2] P. Deitel. Understanding Java 9 Modules. https://www.oracle.com/corporate/features/understanding-java-9-modules.html, 2017.

[3] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. *IEEE*, 2014.

[4] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, 2011.

[5] GCC. Official Documents. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html, 2020.

[6] GCC. Wiki. https://gcc.gnu.org/wiki/cxx-modules, 2020.

[7] N. Ghorbani, J. Garcia, and S. Malek. Detection and repair of architectural inconsistencies in java. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[8] M. Hammad, H. Bagheri, and S. Malek. Determination and enforcement of least-privilege architecture in android. In *IEEE International Conference on Software Architecture*, 2017.

[9] M. Hammad, H. Bagheri, and S. Malek. Deldroid : An automated approach for determination and enforcement of least-privilege architecture in android. *Journal of Systems and Software*, 149(MAR.):83–100, 2019.

[10] ISO/IEC. ISO/IEC 14882:2020 Programming Languages - C++. https://www.iso.org/standard/79358.html, 2020.

[11] T. Lutellier, D. Chollack, J. Garcia, L. Tan, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *37th International Conference on Software Engineering (Software Engineering in Practice). ICSE-SEIP*, 2015.

[12] P. Manadhata and J. M. Wing. Measuring a system's attack surface. *advances in information security*, 2004.

[13] P. K. Manadhata, K. Tan, and R. A. Maxion. An approach to measuring a system's attack surface. 2007.

[14] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[15] OpenJDK. Project Jigsaw. `http://openjdk.java.net/projects/jigsaw/`, 2017.

[16] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 2000.

[17] M. Reinhold. JSR 376: Java Platform Module System. `http://cr.openjdk.java.net/mr/jigsaw/spec/`, 2014.

[18] P. Remy. Easy-encription. `https://github.com/philipperemy/easy-encryption`, 2017.

[19] G. Shahmohammadi, S. Jalili, and S. M. H. Hasheminejad. Identification of system software components using clustering approach. *Journal of Object Technology*, 9(6):77–98, 2010.

[20] K. Sharan. *The Module System*. Java 9 Revealed, 2017.

[21] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, 2000.

[22] O. Welsh. TextBasedAdventure. `https://github.com/OwenWelsh/TextBasedAdventure`, 2013.