Efficient Compilation Using a Program Database

2U－3

Kazushi Kuse
IBM Tokyo Research Laboratory

1.0 Introduction

The long program compilation and linking time is an issue of increasing concern in object-oriented programming languages such as C++. One reason for the problem is that class structures create a lot of dependency. Another is that an object-oriented program refers many header files in order to use class libraries. First, to clarify the issue, we examine actual programming activity in C++. We then describe a method for reducing the compilation and linking time of C++ programs by using a C++ program database [1]. On the basis of this method, we developed a program builder that succeeded in reducing the compilation time of C++ programs by over 80%; in the last two sections, this builder is evaluated and discussed.

2.0 Observations of C++ Program Builds

We made observations of actual builds of a C++ program. One hundred and fifty five successful program builds took place over five days. The types of modifications made during the program builds are classified as follows:

1) Modification of header files
2) Modification of source files
2-1) Addition of new uses of a global name in a function
2-2) Deletion of all uses of a global name from a function
2-3) No modification of the uses of all global names

The proportions of modification types **1)**, **2-1)**, **2-2)**, and **2-3)** were 11.6%, 23.9%, 8.4%, and 56.1%, respectively. Evidently we do not need any new header file information to compile modified functions in the case of **2-2)** or **2-3)**. Together, these account for 64.5% of all modifications. If we evaluate the same value by using a file scope rather a function scope, the proportion increases to 78.7%. This implies that an impact analysis result for header file information can be used in some continuous program builds.

3.0 Program Build Scheme for C++ Using a Program Database

We describe a way of collecting minimal information for program recompilation by using a program database. The database stores static program information, which is populated by the C++ compiler. The build method uses a unit of global declarations such as a class, type, or function definition. Some smaller units of analysis, such as statements or expressions, would be other candidates, but our observations indicated that global-declaration-level analysis is enough to achieve a significant reduction in program build time.

3.1 Use of the Program Database

The main function of our program builder is impact analysis, for which it uses the program database. There is a dilemma: a program build needs the latest program database information, but on the other hand, to get the latest program database, a program build is needed. The following are possible approaches to solving the dilemma:

1. A program build uses the results of the previous impact analysis. First, the builder tries to compile a modified function by using the results of the previous impact analysis, instead of updating the database by compiling the function. If there are any compiler errors caused by insufficient header file information, the builder compiles the program again, using all the header file information, and updates the database. The cost of the trial compilation is small compared with that of a full compilation, since the builder uses only the necessary declarations.

2. Each header file keeps the result of impact analysis, and the impact information is recalculated when the content of the header file is changed. This approach is particularly effective for user-defined classes, because these use standard class libraries. The lifetime of an impact analysis result for header files is longer than that of a result for source files in the previous approach.

3. Using the program database, a simple parser detects global name references in a modified function before the actual impact analysis. There are several possible levels of detection. We could obtain precise reference information by deep analysis, or approximate information by simple analysis.

In our build scheme, we took the first and second approaches, to eliminate the change detection time.

3.2 Impact Analysis for Source File Changes

The following is an overview of the impact analysis.

Step 1. Calculate a set of names directly used by the changed function by retrieving the following entities from the program database:

- Use of class name
- Use of global type name
- Use of global variable name
- Use of macro name
- Function call

Step 2. Find out the following definitions for the names used by scanning the global declarations in the program database:

- Class definition
- Type definition
- Macro definition
- Function definition
- Variable definition

Step 3. Calculate a set of names used in acquired definitions. The above two steps are executed in turn until no new uses or definition names can be found.

3.3 Impact Analysis for Header File Changes

A change of a header file has a strong impact on the program build time, because the file may be included in multiple source files. This is also analyzed on the basis of a global declaration unit. The impact of header file changes is calculated by reversing 3.2. After all the effects on use information have been detected in source files, the program builder checks all the effects on definitions in the way described in the previous section.

4.0 Evaluation

We measured the effectiveness of the program builder by using a sample program, which is an X-Window application of the MotifApp Framework. It has 4,018 LOC in C++, with 65 classes and 179 functions. The program is implemented in 46 separate source files. It takes 143.1 sec to compile the program with a debug option. The linking time for these compiled source files is 4.4 sec. If a shared library is used, this decreases to 1.0 sec.

Table 1 shows a comparison of the code size that will be compiled by the C++ compiler. We selected the smallest and the largest source files from the 46 files, namely, Main.C and Stage.C. We also made a separate file that includes the 'resize' function of Stage.C, in order to evaluate the effectiveness of the function scope analysis.

If one function is included in a file, the reduction ratio is more than 20 to 1. Even in the largest file, which includes eight functions, it is more than 10 to 1. Table 2 shows a comparison of the actual compilation times in each case.

TABLE 1. Code Size of Preprocessed Files (LOC)

File name	No. of funcs.	Original	Without builder	With builder	Ratio
Main.C	1	44	8623	313	27.5
Stage.C	8	241	8842	676	13.1
resize.C	1	63	8706	363	24.0

TABLE 2. Comparison of Compilation Times (sec)

File name	Without builder	With builder	Ratio
Main.C	3.21	0.45	7.1
Stage.C	3.55	0.74	4.8
resize.C *	3.32	0.47	7.1

The compilation speed with the builder is more than four times faster than without it. We calculated each build time by adding the linking time, 1.01 sec. The build speed is more than twice as fast as the original build. We improved the build speed by providing a faster incremental linker that uses code static information in the program database.

5.0 Discussion

Although we do not show the cost of the impact analysis in the program builder, we were able to reduce the number of impact analyses by providing a temporal header file called the 'optimal header file' for source file changes. Our observations showed that in most cases, the use of global names in a function is not changed. Therefore, if the program builder generates an optimal header file that includes minimal declarations for functions or files currently being edited, we can compile the functions or files fast by using this optimal header file while it is effective. The optimal header file for a source file has a longer lifetime than one for a function, but its compilation time is likewise longer.

The optimal header file is also effective for reducing the time needed to update the program database. A programmer populates a program database first, and updates it by replacing old information in changed files with new information, so it is also important to reduce the time needed to update the program database.

Reference

[1] T. Onodera, 'Experience with Representing C++ Program Information in an Object-Oriented Database', OOPSLA'94.