SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(5), 477–485 (MAY 1993)

Reducing Compilation Time by a Compilation Server

TAMIYA ONODERA
IBM Research, Tokyo Research Laboratory, 5–19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

SUMMARY

In language systems that support separate compilation, we often observe that header files are internalized over and over again when the source files that depend on them are compiled. Making a compiler a long-lived server eliminates such redundant processing of header files, thus reducing the compilation time. The paper first describes compilation servers for C-family languages in general, and then a compilation server for our C-based object-oriented language in particular. The performance results of our server show that a compilation server can substantially shorten the compilation time.

KEY WORDS: Compilers Compilation server C-based objects

1. INTRODUCTION

Modern programming systems based on compilers support the notion of *separate compilation*. A program in such systems consists of *source files*, which are separately compiled and linked to form an executable, and *header files*, which supply commonly used declarations. Each time a source file is compiled, a new compiler process is created. During compilation, the process *internalizes* declarations in those header files on which the source file depends, building corresponding data structures, such as symbols and parse trees, within the process.

Different language systems internalize declarations in a header file in different ways. For instance, in a C-based language system, the internalization usually consists of two steps: first, the preprocessor reads the original texts of a source file and of header files specified in it by `#include` directives, and writes out a preprocessed source file; then, the compiler parses the resultant file. In a Modula-2 language system, on the other hand, the internalization involves only a compiler. Some Modula-2 compilers, such as the one described by Foster,[1] internalize declarations of a header file by compiling the original text, whereas others, such as the one described by Gutknecht,[2] do so by reading a *precompiled* version of the header file, because precompiled header files can be more efficiently internalized.

Unfortunately, internal data structures built in one compiler process are never shared by another compiler process in most systems. As a result, a group of compiler processes repeatedly internalizes declarations in header files. Let us consider two

0038–0644/93/050477–09$09.50

Received 30 January 1992
Revised 24 November 1992

typical situations that occur during the development of a program—*massive* compilation and *repetitive* compilation. A massive compilation, in which many source files are compiled in series, is caused when an attempt is made to build an executable after making modifications that influence many of the source files; it sometimes occurs even as a result of a single modification to a header file. Since each header file is ordinarily used in more than one source file, redundant internalization occurs.

More ubiquitous is the problem of repetitive compilation, in which a single source file is captured in the cycle of edit-compile-debug. Each invocation of a compiler involves internalizing the same set of header files, although they are unchanged during this repetitive compilation. Again, redundancy obviously results.

In order to estimate roughly how much redundancy is involved, let us consider a directed graph whose nodes are source and header files. An arc is drawn from a header file to a source file if the latter depends on the former, either directly or else indirectly through another header file. Then, the outdegrees of header files can be regarded as a general indication of how redundant internalization is involved in a massive compilation. Similarly, the indegrees of source files give a general indication of how redundant internalization is entailed in a repetitive compilation. Table I shows these statistics, among other data, for three moderate-sized programs. The first two are C++ programs, whereas the third is written in COB,[3] a language developed at IBM's Tokyo Research Laboratory. COB is also a C-based object-oriented language, and its major features are (1) class interfaces without private members, (2) run-time type descriptors, (3) garbage collection and (4) on-demand internalization of class interfaces. When we collected statistics on the header files, we focused on those containing class interfaces; we did not count system header files such as stdio.h and string.h.

The compilation server is a long-lived compiler process that accepts and handles successive compilation requests from a client. The most important feature is that it can retain internal data structures generated while serving a request and use them to deal with subsequent requests; it requires header files to be internalized only once at most. We can thus expect it to reduce the compilation time in both massive and repetitive compilations.

This paper describes our experience with creating and using a compilation server, in which COB is used as both the source language and the implementation language. The remainder of this paper is organized as follows. Section 2 discusses compilation servers for C-family languages and complications in making them caused by preprocessing in these languages. Section 3 describes a compilation server for COB in which the language's features have allowed us to adopt a cleaner and easier solution.

Table I. Redundancy involved in massive compilation and repetitive compilation

Program	Number of lines	Number of files		Outdegrees of header files		Indegrees of source files	
		Header	Source	Maximum	Average	Maximum	Average
InterViews 2.6	33·9K	95	86	80	8·17	28	9·02
NIH class library 3.0	16·5K	57	52	50	8·49	21	9·31
COB compiler	22·7K	167	143	104	12·33	117	14·40

Section 4 presents the performance results. Section 5 overviews related work, and finally, Section 6 offers some concluding remarks.

2. COMPILATION SERVERS FOR C-FAMILY LANGUAGES

Unlike Modula-2 and Ada, C involves the notion of *preprocessing*. When we make a compilation server for C, this affects the structure and complicates the implementation, as we will discuss below. Although the discussion specifically concerns the C language, it also applies to other languages that are extensions of C, such as C++ and COB, insofar as they rely on preprocessing.

Controlling flexibility

C programmers use #include preprocessor directives in order to specify in a source file the header files whose declarations must be internalized. The text of each header file is processed as if it appeared in place of the #include directive. Since this is simply a form of textual processing, header files in C can supply more than just declarations. Here is a pathological example, showing the contents of three files:

```
% cat foo.h def.c use.c
/* foo.h */
foo

/* def.c */
#include "foo.h"
(int i){ printf("foo(%d) is called",i); }

/* use.c */
main(){
#include "foo.h"
(3);
}
%
```

The two inclusions are not redundant in this case, and neither can be skipped.

The compilation server for C must therefore discriminate between header files containing only declarations and those containing both declarations and other objects, without inspecting the contents. Fortunately, the ANSI C language introduces a #pragma directive to supply implementation-dependent information. We can use this directive to distinguish header files. For instance, we can provide information such as the following and interpret it so that the specified file must always be processed:

```
#pragma include(foo.h)
```

In the rest of this section we discuss a server that incorporates this interpretation, and focus on avoiding redundant processing of #include directives specifying files that are not listed in #pragma directives.

Using a preprocessing compiler

As we mentioned earlier, the process of internalization is divided into two steps, which are carried out by the preprocessor and the compiler, respectively. In some implementations, the preprocessor and the compiler are actually separate programs, and simply making the compiler a server can at best eliminate only the second half of redundant internalization; we will still see header files preprocessed over and over.

In other implementations, a single compiler program does both the preprocessing and the compiling. By building a server from such a preprocessing compiler, we can fully avoid redundant internalization of header files. We focus on this kind of server in the rest of this section.

Handling inclusion directives

In order to avoid redundant internalization, a preprocessing server can handle an inclusion directive in the following way. When the server encounters an inclusion directive for a header file for the first time, it creates a flag for the header file, and then actually preprocesses and compiles the header file. When the server encounters a subsequent inclusion directive for the same header file, the server finds that the flag already exists, and simply ignores the directive.

This approach is based on the premise that a header file is inserted in the same way into every source file containing an inclusion directive for the header file. Unfortunately, the preprocessor conditional directives, such as #ifdef and #ifundef, make it possible to write pathological programs in which the above premise does not hold. In order to deal with such pathological programs, the server needs to take account of *free* occurrences of names in a header file; as in λ-terms, the occurrence of a name is defined to be free in a header file if and only if the header files does not have any definition directive for the name before the occurrence. We must then use instead of a simple flag a more elaborate data structure, namely, a list of pairs of name and definition, which we call an *n-list*.

A modified algorithm is as follows. When the server encounters an inclusion directive for a header file for the first time, it creates for the header file an empty n-list. Each time the server finds a free occurrence of a name while preprocessing the header file, it appends the name and its current definition to the n-list. When the server encounters a subsequent inclusion directive for the same header file, the server compares the previous and current definitions for each name stored in the n-list. If the definitions are the same for every name, the server ignores the directive. Otherwise, it discards internal data structures resulting from the header file, makes the n-list empty, and internalizes the header file again. During preprocessing in the internalization, the server newly constructs an n-list.

3. COMPILATION SERVER FOR COB

In this section we describe a compilation server for COB. Since COB is an extension of C, making a server for COB involves the difficulties mentioned earlier. Although we could get around them in the way discussed in Section 2, the features supported by COB have enabled us to take a very different approach. We also discuss the storage management method used in our compilation server.

Concentrating on class interfaces

Since COB is an object-oriented language, COB programs are organized around classes, which are defined through interfaces and implementations. Primarily, the header files contain class interfaces, and the source files contain class implementations. Since COB is compatible with C, a class implementation can rely on functions written in C, and a program contains C functions. Thus, some header files containing C declarations are still used. A header file is not allowed to contain both class interfaces and C declarations. Header files used in COB are therefore divided into two groups, *interface files* containing class interfaces and *C declaration files* containing C declarations.

The language does not allow preprocessor directives to appear in interface files, but of course allows them to appear in C declaration files and source files. Accordingly, the COB compilation server attempts to avoid redundant internalization of interface files, but not of C declaration files. Thus, we do not have to use a preprocessing compiler or to handle inclusion directives in a complicated manner, although we do not avoid all redundancy.

The above-mentioned restriction imposed upon interface files is not severe. C programmers often use definition directives and inclusion directives, and COB programmers would also want to use them even in interface files. Fortunately, two language features have made them practically unnecessary. First, common members (static members in C++ terminology) declared with the `const` type qualifier are treated in exactly the same way as names defined in `#define` directives if they have initializers and the types are arithmetic.

Secondly, COB supports *on-demand internalization* of interface files, which eliminates the need to write inclusion directives for class interfaces and to keep inclusion directives consistent with the actual uses of classes. When a compiler process encounters the use of an undefined class during the compilation of a source file, it attempts to internalize the class interface by finding* and compiling an appropriate interface file; the compiler process suspends the compilation of the source file, starts compiling the interface file, and, upon completion, resumes the suspended compilation. In most cases, this becomes recursive; during the compilation of an interface file that is found, the compiler process may encounter another undefined class, at which time the compiler suspends the current compilation and initiates a new compilation.

Storage management

A compilation server usually runs for a long time, and is therefore written in such a way that it uses dynamic storage for internal data structures such as symbols and parse trees. While a server is running, garbage accumulates within it as a result of the temporary use of dynamic storage and of the cancellation of previously built objects that have become obsolete. Since these structures form a complicated network, it is best to rely on a garbage collector to manage dynamic storage. Our compilation server is written in COB itself, and the language supports automatic storage reclamation by a generational copying collector.[4]

---

* As the description implies, we have to supply a mapping between the names of possibly undefined classes and the names of corresponding interface files. This is very easy, since we have a default rule stating that the interface of a class is contained in the interface file whose base name is the same as the class name.

When the server has accepted a request for compilation of a source file, it initiates compilation and creates internal objects. Some objects result from internalization of C declaration files and interface files, and others from compilation of the original contents of the source files. Only the internal objects from interface files survive after the compilation. Other internal objects become garbage immediately after the server has finished the compilation.

When an interface file *H* is updated, the internal objects from *H* become obsolete and must be converted into garbage. For this purpose, a request for cancellation of an interface file is provided. Upon receipt of such a request for *H*, the server turns into garbage *all* the obsolete internal objects. That is, the server converts into garbage the internal objects from *H*, those from the interface files that depend on *H*, and so on. The current version simply cancels all the interface files on the basis of the dependency relationship, although Tichy[5] proposes a smarter approach.

4. PERFORMANCE RESULTS

We have measured our compilation server on a NeXT Mach 2.5, whose real memory is 16 MB and whose processor is a 25 MHz Motorola 68030. Our COB compiler is realized as a front-end to C. That is, each source file of a COB program is first preprocessed by a conventional C preprocessor, the output is then translated by our COB compiler into a C source file, and the result is compiled by a conventional C compiler. The following timings do not include the time taken by the C compiler. Note also that our server does not avoid redundant internalization of such header files as stdio.h.

First, we compiled the entire source code of the compiler itself, which amounts to 140 source files, with a total of almost 21,800 lines, and 168 interface files, with a total of 2438 lines. We compiled them both by running the compiler in a separate-compilation mode and by running the compiler in a server mode. The results are shown in Table II. Although the garbage collector was invoked as many as 168 times, the user time, system time, and real time were all better when a compilation server was used. We turned off the garbage collection during the separate compilation, or rather we could say that we simply relied on the ‘garbage collection’ that the operating system performs when it kills a process; the times in separate compilation would have been much worse with the garbage collection turned on. We may conclude that the compilation server substantially reduces the total recompilation time in massive compilation.

The time spent on garbage collection is 501 seconds. This seems quite a lot of time, but is actually a little less than the time that would be spent if we explicitly

Table II. Conventional compiler and compilation server using a generational copying collector

Compilation method	User time (s)	System time (s)	Real time (m)	Utilization (%)	Number of garbage collections
Separate compilation	2149·4	184·7	45·16	85	—
Compilation server	1544·6	67·6	27·51	96	168

managed dynamic storage by calling the free() function at appropriate places; we have estimated the time from the numbers of live and garbage objects; see Reference 4 for details. At any rate, we have to pay a substantial cost to manage dynamic storage, whether automatically or manually, for programs running very long.

The amount of dynamic storage used in this experiment was 7·3 MB at peak and 4·4 MB when all the compilations had been finished; the latter figure approximates the total size of internal objects from 168 interface files. One might think that retaining all the data structures built would lead to a slower performance in terms of the working set theory, and that it would be preferable to keep a minimal set of data structures by using a separate compiler. However, as we have seen, given the size of real memory in recent workstations, the compilation server outperforms the separate compiler even for a medium-sized program containing tens of thousands of lines.

The second experiment is related to a repetitive compilation. Table III shows the amounts of time that a compilation server spent handling the first request to compile a source file and the second request to compile the same file. The times shown in the last two columns are real times. If a conventional compiler were used, the second invocation of the compiler would take almost the same amount of time as the first one. This table shows that the compilation server is also effective for reducing the compilation time in repetitive compilation.

5. RELATED WORK

Gutknecht[2] and Fyfe *et al.*[6] propose precompilation of header files, which converts them into representations that can be more efficiently internalized. When compilation of a source file requires a header file to be internalized, each of the compilers they describe loads the precompiled version, if one exists. Otherwise, it reads and compiles the original text of the header file, and generates the precompiled version as a side-effect. (Gutknecht and Fyfe *et al.* take two different approaches to management of the precompiled versions). Some PC-based C++ compilers, such as Borland C++,[7] support precompilation of the header files.

Here we compare a precompiling compiler and a compilation server. We first divide the task of internalizing a header file into disk reading and conversion into internal objects. Whichever of the two methods is used, the compiler performs the first internalization of a header file by reading the original text and converting the source text into internal objects. Let us denote the lengths of time spent on the reading and on the conversion by T_r and T_c, respectively. A subsequent internalization of the header file involves

Table III. Repetitive compilation by compilation server

Source file	Number of lines	Interface files		Compilation time (s)	
		Number of files	Total lines	1st	2nd
AddExp.cob	58	9	252	20	9
main.cob	191	33	751	40	12
CSpace.cob	732	42	1033	69	35

1. In a precompiling compiler, reading the precompiled version (which takes less time than T_r) and converting the compact representation into internal objects (which takes less time than T_c).
2. In a compilation server, nothing. Any subsequent internalization is free.

In addition, precompilation entails the run-time overhead of writing out precompiled versions of header files in their first internalizations, and creates the programming overhead of having to manage both the original and precompiled versions.

Fyfe *et al.*[6] propose a compile server (as they call it) for C, which they use primarily in conjunction with the debugger. The debugger can share functions for parsing and lexical analysis with the compiler, and does not need to have functions of its own. Their compile server can also be used for our purpose, namely for reducing the compilation time. However, the difficulties discussed in Section 2 must be addressed before this can be realized.

Tichy[5] proposes a method, called *smart recompilation*, for reducing the set of modules that must be recompiled after a change. This also helps reduce the compilation time, and is orthogonal to our approach. It is effective in massive compilation, but not in repetitive compilation.

Finally, Atkinson *et al.*[8] and Jordan[9] also touch on the notion of a compilation server, but do not present enough details to allow us to compare it with our approach.

6. CONCLUDING REMARKS

We have shown that a compilation server is effective in reducing the compilation time both in massive compilation and in repetitive compilation. We believe that the idea of retaining a substantial number of objects in virtual memory has become feasible only in today's workstations, which have large real memories.

As we have seen, making a compilation server is easier for languages such as Modula-2 and Ada, than for C-family languages. Macro processing is a major obstacle. As a matter of fact, it complicates the implementation of many programming tools. For instance, it is hard to implement a debugger that can operate on original source files instead of macro-expanded source files, and a browser that can handle pathological cases where header files included in different source files are processed differently. We believe that it is better to shift to a Modula-like approach to internalization such as we have taken with COB.

As many researchers suggest, a compilation server is not simply a tool for reducing the compilation time, but can function as a central tool in a programming environment. This is because the symbols and parse trees kept in the server process represent almost all the aspects of the source files compiled. By defining an appropriate application interface to the server, we expect to be able to build around the server such tools as a browser and a debugger in an effective manner. To achieve this, the server has to retain many more objects than described in Section 3, and will thus impose a much heavier demand on real memory. However, the rate at which the capabilities of dedicated personal workstations are progressing is so fast that we believe a server-centred programming environment that can deal with medium-sized programs is quite feasible.

ACKNOWLEDGEMENTS

Fumihiko Kitayama collected statistics on C++ programs. Thanks also to Tsutomu Kamimura, Kazushi Kuse, and other colleagues at Tokyo Research Laboratory for participating in discussions on this work.

REFERENCES

1. D. G. Foster, 'Separate compilation in a Modula-2 compiler', *Software–Practice and Experience* **16**, 101–106 (1986).
2. J. Gutknecht, 'Separate compilation in Modula-2: an approach to efficient symbol files', *IEEE Software*, **3**, (11), 29–38 (1986).
3. T. Onodera and T. Kamimura, 'COB language manual', IBM Research, Tokyo Research Laboratory, March 1990.
4. T. Onodera, 'Generational and conservative copying collector for C-based objects', *Research Report RT0070*, IBM Research, Tokyo Research Laboratory, December 1991.
5. W. F. Tichy and M. C. Baker, 'Smart recompilation', *Proc Twelfth POPL*, 1985, pp. 236–244.
6. A. Fyfe, I. Soleimanipour and V. Tatkar, 'Compiling from saved state: fast incremental compilation with traditional UNIX compilers', *Proc. Winter 1991 USENIX Conference*, 1991, pp. 161–171.
7. Borland C++ User's Guide, Borland International, 1992.
8. R. Atkinson, A. Demers, C. Hauser, C. Jacobi, P. Kessler and M. Weiser, 'Experiences creating a portable cedar', *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989, pp. 322–329.
9. M. Jordan, 'An extensible programming environment for Modula-3', *SIGSOFT Software Engineering Notes*, **15**, (6), 66–76 (1990).