CCC: A Caching Compiler for C

Brian Koehler and R. Nigel Horspool[*]

Department of Computer Science

University of Victoria, P.O. Box 3055

Victoria, BC, Canada V8W 3P6

14 December 1995

Abstract

C compilers in production environments often have to reprocess the same header files. CCC is a caching C compiler which maintains a database of partially processed header files so that repetitive work may be eliminated.

Keywords: Compilers, Compilation server, C.

1 Introduction

A large C program is usually composed of many source code files. Some files contain implementation code consisting of function declarations and data definitions; these files have names with a '.c' suffix and are commonly called '.c' files (pronounced as "dot see files"). Other files serve the purpose of defining interfaces between program modules or between the program and previously compiled library functions. These files normally contain function prototypes, and data type definitions. They have names with a '.h' suffix and are commonly called header files or '.h' files ("dot aitch files").

A typical '.c' file contains several `#include` directives which each specify a header file. A `#include` directive causes the compiler to process the header file as though the contents of the file had been inserted into the '.c' file instead of that `#include` line. Amongst other things, the definitions inside header files allow the compiler to generate code for calls to functions that are implemented in other '.c' files.

Compilers, particularly C compilers, perform much redundant work through repeated processing of the same unchanged files. We can easily avoid re-compiling an unchanged '.c' file by using a tool like *make* which checks the timestamp associated with the file. However, it is much harder to avoid re-processing an unchanged '.h' file. There are two basic situations where repeated processing occurs. The first situation occurs if we compile two different '.c' files that happen to include the same header file. The compiler will be invoked twice, and

[*]E-mail: `nigelh@csc.uvic.ca`

these two invocations will repeat the same processing of the text in the header file. The second situation occurs when we make a modification to a '.c' file. When we recompile the file after making our change, the chances are that the same header files will be included as in earlier compilations of the '.c' file and that these header files have not been changed.

A general solution to the problem of avoiding repeated work involves saving the results of processing the header file, and re-using the saved result in future compilations. Our approach uses a *compilation server*. The server is continuously available and provides a C compilation service to C compiler clients. The server caches the results from processing a header file. If a subsequent compilation request involves the same header file, then the saved results may be retrieved from the cache.

In the following sections of the paper, we will describe related work in the area. Then we will explain our approach, we will discuss its implementation in the CCC compiler, and we will conclude with some experimental results that show the effectiveness of caching.

2 Background and Previous Work

In a typical C compiler, source code passes through various processing stages. The first stage is lexical analysis which tokenizes the source code into its lexical units. The lexical analysis is intimately coupled with the operation of the *C preprocessor* which supports simple macro definitions, conditional compilation directives, as well as the `#include` directive. After lexical analysis, the compiler performs syntactic analysis, semantic analysis, and object code generation.

There is no special treatment for header files. They are subject to the same processing as '.c' files. However, although there is no language requirement to do so, header files almost always contain only declarations. A header file's entire effect on a compiler may be summarized as its effect on the preprocessor (e.g. by declaring new macros) and its effect on the compiler's symbol table. Figure 1 shows the overall structure of a typical C compiler as it pertains to our problem.

A variety of approaches for reducing or eliminating unnecessary reprocessing of a header file has been reported. We can identify four main schemes, as follows.

1. The compiler can cache a tokenized version of the header file. In principle, either the token sequence before pre-processing or the sequence after pre-processing could be saved. The latter choice, however, eliminates more work in subsequent compilations.

2. The header file can be viewed as little more than a series of declarations for various program identifiers. Litman's approach is to construct and save an index table which records which symbol is declared where in the header file [4]. If the same header file is subsequently included in another compilation, the index table is used to look up the locations of identifiers' declarations. The text of these declarations are lexically analyzed and reprocessed as needed. Litman's scheme works best if only a small fraction of the identifiers declared in a header file are actually referenced. Such a situation is very common.

3. The state of the compiler's symbol table is cached immediately after the header file is processed. This scheme works easily and reliably only in two situations. The first

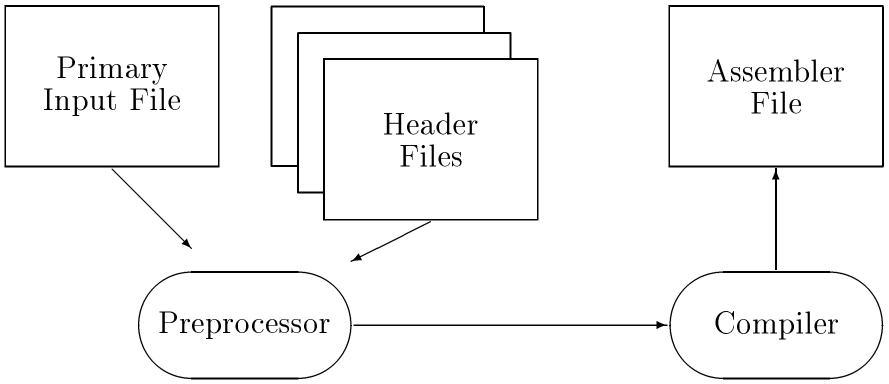

Figure 1: Structure of a Typical C Compiler

situation occurs when the text of the header file is the only text (other than white space and comments) to have been processed by the compiler. In other words, only one header file can be handled in this manner and the `#include` directive for the header file must be the first action in the '.c' file. The Symantec Think C/C++ compiler on Macintosh computers uses this scheme. A very large header file containing declarations for interfaces to the MacOS operating system is available in a pre-compiled manner [6].

The second situation is a generalization of the first. If one compilation saves the compiler's symbol table after a sequence of `#include` directives, and if a subsequent compilation has an identical sequence of `#include` directives, then the cached symbol table can be re-used. The Borland C/C++ compiler uses such a scheme.[2]

4. Onodera described a compilation server for the COB language [5]. COB is a C-based object-oriented programming language. COB source code is provided in three kinds of files: '.c' files, interface files which contain only **class** declarations, and declaration files which play a role similar to header files in C. The COB compilation server maintains an in-memory cache of previously compiled interface files. COB declaration files are not cached.

   Our approach in CCC is closer to Onodera's than any of the others.

3 The C Compilation Server

When a user types the command to perform a C compilation, the command runs a small client program which passes the command-line information onto a C compilation server.

```
#ifdef DEBUG
void debugPrint(char *message);
#define TRACE(m) debugPrint(m);
#undef FAST
#else
#define TRACE(x)   (0)
extern double pi;
#endif
extern void solveFermatEquation(int n);
```

Figure 2: The Header File "example.h"

As with any server daemon, it is permanently alive but is normally in an inactive state waiting to receive a request for service. When it receives a request, it performs the requested compilation and returns the result (an object file) to the client program. Our C compilation server, CCC, is a modified version of the *lcc* compiler [3].

When a header file is processed, that header file may cause two kinds of changes to the state of the compiler.

1. There may be changes to the state of the preprocessor. Our compilation server handles only two kinds of changes: new macro definitions and deletions of previously defined macros. If a header file should have any other effect on the state of the preprocessor, the header file would not be cached.[1]

2. There may be changes to the combined state of subsequent compiler phases. Our compilation server handles only changes that take the form of additions to the compiler's symbol table. If the header file should cause any other effects, the header file would again be considered ineligible for caching.[2]

Our compilation server records information about both kinds of effects that a header file produces. We will now elaborate on exactly what information is recorded.

3.1 Recording Changes to the Preprocessor State

The information related to the change in state of the preprocessor is easier to describe by means of an example. Suppose that the '.c' file contains the lines of text that are shown in Figure 3. and suppose that the header file, *example.h*, contains the text shown in Figure 2. We further suppose that no identifiers, other than `DEBUG`, that are used in *example.h* have been defined as macros.

For the *example.h* header file, the following information is retained in the in-memory cache after preprocessing.

[1]For example, one such effect occurs if a `#ifdef` directive appears in the header file and no matching `#endif` directive is provided in the header file.

[2]For example, a header file could contain executable C statements, and these would change the state of the code generation phase of the compiler.

```
/* preceding lines are omitted */

#define DEBUG
#include "example.h"

/* following lines are omitted */
```

Figure 3: The Implementation File “example.c”

Exposed (free) identifiers:

{ `DEBUG`, `void`, `debugPrint`, `char`, `message`, `extern`, `solveFermatEquation`, `int`, `n` }

Note that *pi*, for example, is not an exposed identifier because it appears only in a section of the file that is by-passed by a conditional compilation directive. Further note that C keywords such as `extern` are considered to be identifiers by the preprocessor.[3]

Deleted macros:

{ `FAST` }

Additional or changed macros:

{ < `TRACE`, 1, `debugPrint` (@1) > }

Each macro is recorded as a triple composed of the name, the number of parameters, and the body of the macro. The identifiers used as formal macro parameters are not significant. The parameters are therefore replaced by special tokens: `@1` represents the first parameter, `@2` represents the second, and so on.

The cached information for the *example.h* header file should be reusable in another compilation if the `DEBUG` macro is again defined before the header file is included. However, we need to be cautious. If, say, the identifier *debugPrint* were to be defined as a macro in one compilation and not in the other, then the token sequences produced by the header file in the two contexts would be different.

A sufficient condition to guarantee that a cached header file can be re-used is that all exposed indentifiers must have identical definitions. Therefore, we attach a list of the definitions for the exposed identifiers to a cached header file so that we can verify the validity of the substitution. For each exposed identifer we record whether or not it was defined and, if it was, we record the number of parameters and the body. For our example, the definitions would be as shown below.

{ < `DEBUG`, *Defined*, 0, λ >,
< `void`, *NotDefined* >,
< `debugPrint`, *NotDefined* >,
< `char`, *NotDefined* >,
...
}

[3]In principle, one could define *extern* as a macro. Needless to say, such practice is to be deplored.

File *types.h*:

```
typedef long time_t;
```

File *time.h*:

```
time_t clock(time_t *arg);
struct usage *getusage(void);
```

File *usage.h*:

```
struct usage {
        time_t user_time, system_time;
};
```

Figure 4: Dependencies Between Symbol Table Entries

The λ symbol denotes an empty macro body.

If the test that exposed identifiers have identical definitions succeeds, we can apply the changes to the processor state that have been recorded. That is, we add or replace some macro definitions, and we delete some others. Then the cached text of the header file can be forwarded to the next phase of the CCC compiler.

3.2 Recording Changes to the Compiler Symbol Table

In addition to eliminating the preprocessing work when we use a cached header file, we attempt to eliminate much more work. After preprocessing, a header file normally contains only declarations – such as type definitions, external declarations of variables, and function prototypes. Such declarations do not cause the compiler to emit any code; their only effect should be to cause new entries to be added to the compiler's symbol table. We therefore attempt to cache the symbol table additions that the header file introduces. If a header file should contain a C language construct that causes code to be emitted (such as a declaration for a variable with an initializer) or that changes the state of the compiler in some manner other than adding symbol table entries, then symbol table caching is suppressed. We further require that the `#include` directive not be nested inside any C language construct (such as inside a function). This means that only entries at the global scope are added to the symbol table. (Some definitions, such as those for *struct* types contain nested definitions however.)

As a pragmatic issue, there is little point in caching new symbol table entries from a header file unless these entries can be added to the compiler's symbol table efficiently. This observation has implications for the implementation of the symbol table and for how header files should be handled. How should we process symbol table entries which refer to other symbol table entries? And what if those other symbol table entries were created by other header files? An example scenario is shown in Figure 4. In our example, the symbol table entry for `clock` will contain a reference to the entry for `time_t`. The handling of the structure tag `usage` is interesting. A forward reference to the `struct usage` type, as exemplified in

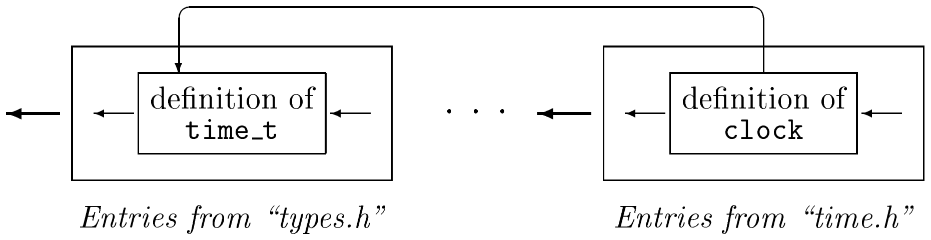

Figure 5: Symbol Table Structure in the Server's Cache

the "time.h" header file, is permissible in C. A typical C compiler would add an entry for the `usage` structure tag to its symbol table when the `getusage` declaration is processed. However, the structure type will be flagged as incomplete. Later, when the declaration of the `struct usage` type is processed, the incomplete symbol table entry will be accessed and completed. This causes both forward and backward dependencies between the header files.

In CCC, the compilation server maintains a large symbol table that combines the symbol entries generated by all the cached header files. The entries are threaded in order of their declaration in particular compilations,[4] and entries may refer to each other (such as when the `clock` entry contains a reference to the `time_t` entry.) Symbol table entries generated by a particular header file will form a consecutive block of entries in a thread. After processing the header files of Figure 4, the server's symbol table will have the structure that is sketched in Figure 5.

When a new compilation is started, CCC will start a new thread of symbol table entries. If a header file such as "time.h" in our example is to be included in the new compilation and if the checks for validity are satisfied, we would like to transfer the block of symbol table entries from CCC's cache to the current symbol table thread. This involves no more than breaking the link from the first symbol table entry in the "time.h" block to its predecessor, and replacing it with a pointer to what had been the latest entry in the current compilation thread.[5] However, the cross-reference from the `clock` entry to the `time_t` entry represents a danger. Perhaps the current compilation is not using the same definition for `time_t`? Perhaps the program is erroneous and there is no definition for `time_t` at all?

To avoid all such problems, we impose a context requirement on the re-use of symbol table entries. For each header file, we record which other header files it directly depends on. That is, if a symbol table entry created by a declaration in header file "A.h" contains a reference to an entry created by header file "B.h", then "A.h" directly depends on "B.h". We will use the cached symbol table entries for a header file "A.h" only if valid cached blocks of symbol table entries for all header files that "A.h" depends on have also been included in the current compilation. A cached block for a header file "B.h" is *valid* only if the references in "A.h" to entries created for declarations in "B.h" are to symbol table entries in the *same* block as were previously included in the current compilation.

[4]In fact, there are many threads because we use hashing to speed look-ups, and each hash bucket has its own thread. We ignore this detail to simplify the explanation.

[5]Our hash table implementation may actually require us to replace several links.

Although it is easy to construct examples where our restrictions are unnecessarily stringent, our restrictions are easy to implement. Any form of restriction that requires elaborate compile-time checking would diminish the benefits of caching. If our checks fail, and the cached symbol table entries for a header file cannot be re-used, the cached preprocessed text of the header file is simply forwarded to the compiler for analysis.

At the end of compilation, any groups of symbol tables entries for header files (or versions of header files) are de-linked from the symbol table and linked into the server's cached symbol table.

3.3 Cache Management

The compilation server maintains all its information in memory. Even if the operating system implements virtual memory, there is a clear benefit to be obtained from purging information for rarely used or superseded versions of header files. Our compiler uses a simple strategy of caching a fixed number of header files. When a new header file is a candidate for caching, the least recently used cached header file is discarded. With this strategy, only frequently used header files are retained in the cache.

4 Lazy Declaration Checking

If the program being compiled is erroneous, there may be duplicate or inconsistent declarations in the header file. In the CCC compiler, the cached symbol table entries for a header file are simply appended to the symbol table for the current compilation without any individual checking.

It would be possible to validate each symbol table entry after loading a block of entries from the cache.[6] However, we have instead chosen to defer such checking until the latest possible moment – in the hope that many of these checks do not need to be performed.

Our new symbol table entries are flagged to indicate that they require validation. When a symbol table entry is subsequently looked up for the first time, the flag will cause the entry to be validated (and the flag is cleared).

Our lazy strategy does mean that some errors can go undetected. As a simple example, suppose that a '.c' file includes both of the header files "a.h" and "b.h". Now suppose that "a.h" contains a declaration

```
struct foo {
    float a;
};
```

and suppose that "b.h" contains the conflicting declaration

```
struct foo {
    int b;
};
```

[6]A compiler option to force this full and immediate validation should probably be provided.

If symbol table entries for "b.h" are obtained from the cache, the conflict will not be immediately detected. If the program being compiled turns out to contain no uses of the `struct foo` datatype, the error will never be discovered.

Although the failure to discover such errors violates the ANSI standard for C compilers [1], we consider that conflicts involving unused symbol table entries are benign. Should, for example, the following statement be reported as an error?

```
        if (0) { int k = 1 / 0; }
```

Division by zero is an error, but the expression that performs the division is never evaluated. We feel that the error with the declaration of the `struct foo` type above has a similar nature.

As suggested above, lazy checking could be disabled by a compilation option if strict checking for conformance to ANSI C standards is desired. The cost for strict checking is a little loss in performance.

5 Experimental Results

The CCC compiler was evaluated using a variety of C source code files from a variety of programs, which included a C preprocessor, the lcc C compiler, the Emacs editor, the Gnuplot plotting package, an image display utility and two flight simulator programs. There were 253 '.c' files in total.

A basic premise of our caching approach is that many more identifiers are declared than are actually used. Our measurements have validated this assumption. We discovered that the proportion of macro identifiers which were subsequently used was only 7%. Of the other identifiers (variable names, etc.), we discovered that 12% were used. This latter figure is artificially high because many identifiers were used only in the declarations of other identifiers, which were themselves unused.

The use of caching imposes an overhead on the compilation time when a header file is processed for the first time. In our CCC implementation, the time required for preprocessing increased by 2%, and compilation time (excluding preprocessing) increased by 10%. Overall, there is a 5% increase in execution time.

If a '.c' file is compiled twice in a row (a common occurrence during the development and debugging of a program), the second compilation requires only 38% of the time of the first compilation, on average. The amount saved depends, of course, on the nature of the C source file being compiled. Figure 6 shows a histogram which displays the execution time savings when each of our 253 '.c' files was compiled a second time.

If a single large program is being compiled, many of the constituent '.c' files will include the same header files. This observation is particularly true of X-windows application programs where many large X header files are typically used. When all the files in a program are compiled in one large batch, the CCC compiler requires 59% of the execution time needed when caching is disabled. Again, different programs yield different savings. Figure 7 shows a histogram displaying the execution time usage.

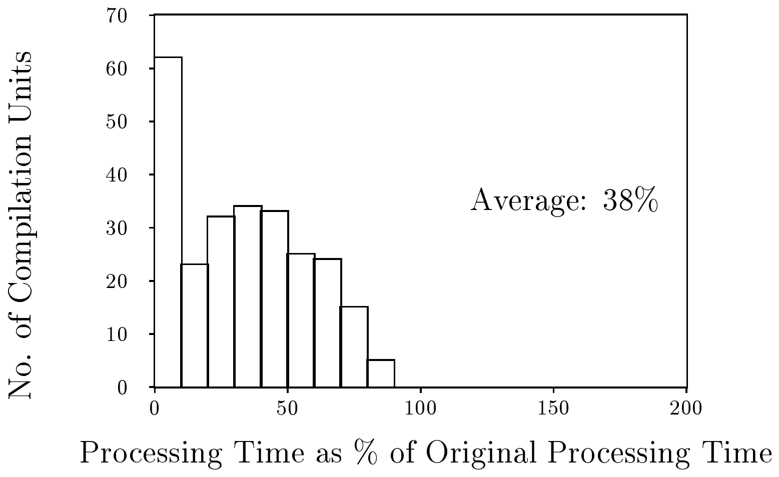

Figure 6: Execution Time Usage for Repetitive Compilation

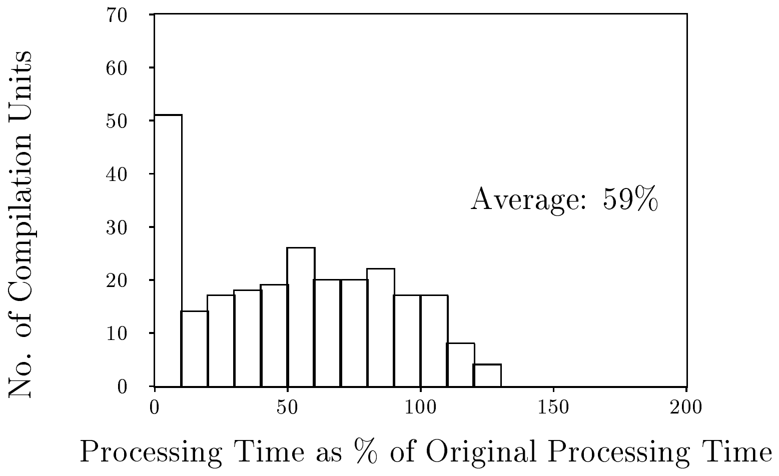

Figure 7: Execution Time Usage for Massive Compilation

6 Conclusions

We believe that we have proved the viability of the compilation server approach for C compilers. More work would be required, however, before CCC could be released as a production compiler. In particular, CCC should handle multiple users. This raises several issues, the most important being protection. Header files belonging to one user should not be accessible to other users unless permitted by the access permissions associated with the files.

Acknowledgement

Financial support for this work was provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] American National Standard for Information Systems – Programming Language – C, X3.159-1989.

[2] Borland International, *Borland C++ User's Guide*, Scotts Valley, CA, 1993.

[3] C.W. Fraser and D.R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin-Cummings, 1995.

[4] A. Litman, "An Implementation of Precompiled Headers", Software–Practice and Experience, 23,5 (1993), pp. 341-350.

[5] T. Onodera, "Reducing Compilation Time by a Compilation Server", Software–Practice and Experience, 23, 3 (1993), pp. 477-485.

[6] Symantec Corporation, *THINK C User's Guide*, Cupertino, CA, 1994.