`Mincpp`: Efficient, Portable Compilation of C Programs

George Moberly

B.S., Boston University

A thesis submitted to the faculty
of the Oregon Graduate Institute of Science & Technology
in partial fulfillment of the requirements for the degree
Master of Science in
Computer Science

April, 1994

The thesis "Mincpp: Efficient, Portable Compilation of C Programs" by George Moberly has been examined and approved by the following Examination Committee:

Tim Sheard, Ph.D.
Assistant Professor
Thesis Research Advisor

Steve Otto, Ph.D.
Assistant Professor

Acknowledgment

This thesis and supporting degree program was made possible only with the financial, intellectual, and emotional support of many corporations, institutions, and individuals. Past and present employers Digital Equipment Corporation, Mentor Graphics Corporation, and SpaceLabs Medical, Incorporated generously paid for the vast majority of the educational expenses involved, and have actively supported my work. Degree credit was transferred from Tufts University, The University of Washington, and Oregon State University. The following individuals shared generously of their time to discuss issues with me: Phil Brooks and Mike Gresham of Mentor Graphics, Bill Bregar of Oregon State University, Aki Fujimura of Pure Software, Greg Kostal, Jeff Small, and of course Tim Sheard and Steve Otto of OGI. Finally, my wife Julie deserves profound credit and thanks for not letting me give up when it would have been easier to have done so, and for unwavering belief and support of this effort.

Table of Contents

TABLE OF CONTENTS IV

LIST OF TABLES VIII

LIST OF FIGURES IX

ABSTRACT X

1. INTRODUCTION/PROBLEM STATEMENT 1

1.1 HELLO, WORLD 2
1.2 NOVA 4
1.3 THESIS PROBLEMS 6
  1.3.1 High Compile Times 6
  1.3.2 Too Many Builds 6
  1.3.3 Too Few Builds 7
1.4 CHAPTER SUMMARIES 7

2. RELATED WORK 9

2.1 OVERVIEW 9
2.2 SYSTEM DESCRIPTIONS 11
  2.2.1 Standard UNIX Compilers: cc, acc and cfront 11
  2.2.2 Cfront/cback 13
  2.2.3 Centerline ObjectCenter 15
  2.2.4 Lucid Energize 18
  2.2.5 Microsoft C/C++ 20
  2.2.6 Borland C++ 20
  2.2.7 IBM Tokyo Research Compile Server 20
  2.2.8 NeXT Smart Preprocessor 20
2.3 DISCUSSION 21

3. MINCPP 22

3.1 Implementation Rationale 22
3.2 Symbol Table Basics 23
3.3 Scanner 25
3.4 Parser Basics 26
3.5 A Single Simple Declaration 28
3.6 Symbol Table: Part 2 31
3.7 External Declarations 32
3.7.1 Simple Declarations *33*
3.7.2 Function Prototypes *35*
3.7.3 Enumerations *37*
3.7.4 Structures and Unions *38*
3.7.5 Typedef Declarations *40*
3.8 Function Definitions 41
3.8.1 Old-Style Parameters *41*
3.8.2 ANSI-Style Parameters *42*
3.8.3 Local Declarations *43*
3.8.4 Referencing *44*
3.9 Save and Replay 44
3.9.1 Save *45*
3.9.2 Replay *45*
3.10 Using Mincpp 46
3.11 Implementation Summary 47

4. RESULTS 48

4.1 Benchmark Descriptions 48
4.1.1 Small Benchmarks *49*
4.1.2 Large Benchmarks *49*
4.2 Directory Structure 50
4.3 Header File Structures 50
4.4 Nocom 58
4.5 Makefiles and Build Scripts 58
4.6 Collection Methodology and Tools 65
4.7 Result Data 66
4.7.1 File Sizes *66*
4.7.2 Cpp and Mincpp Times *69*
4.7.3 Compile Times *71*

4.7.4 Object File Sizes *74*

4.7.5 Linker and Binary Results *76*

4.7.6 Overall Compile Times *79*

4.7.7 Compiler Drivers *83*

4.8 RESULTS SUMMARY 86

5. FUTURE WORK 87

5.1 OVERVIEW 87

5.2 MINCPP SYSTEM 88

5.2.1 Combined Parser *88*

5.2.2 Parse Avoidance and Trace Cache *90*

5.2.3 Build Avoidance and .min.i Cache *92*

5.2.4 The Hub *92*

5.2.5 Multi-Threaded Compile Server *93*

5.3 ANCILLARY TOOLS 93

5.3.1 Minimal #include Generator *93*

5.3.2 "Presto" Syntax Checker *94*

5.3.3 "Dead" Text Eliminator *94*

5.3.4 File Architect *95*

5.3.5 Dependency Manager *95*

5.4 C++ 96

5.5 AGGRESSIVE MINIMIZATIONS 96

5.6 SIDE EFFECT MANAGEMENT 97

5.6.1 Compiler/Debugger *97*

5.6.2 ANSI "Implementation-Defined Behavior" *97*

5.6.3 Extern Objects *98*

5.6.4 Syntax/Semantic Compiler Errors *100*

6. CONCLUSION 101

6.1 KEY BENEFITS DEMONSTRATED 101

6.1.1 Reduced Compile Times *101*

6.1.2 Compiler, Operating System, and Platform Independence *101*

6.1.3 Reduces Object File Sizes *102*

6.1.4 Build Avoidance and make Dependency Leaks *102*

6.2 KEY BENEFITS SUGGESTED 102

6.2.1 C++ Static Initialization *103*

6.2.2 Network Traffic *103*
6.2.3 Point Development Tools *103*
6.2.4 Architecture and Metric Tools *103*
6.3 CLOSING REMARKS 104
REFERENCES **105**
BIOGRAPHICAL NOTE **108**

List of Tables

TABLE 2.1 COMPILATION TECHNIQUE SUMMARY 9

TABLE 4.1 TRANSLATION RULES 62

TABLE 4.2 BENCHMARK FILE SIZES 66

TABLE 4.3 CPP/MINCPP TIMINGS 69

TABLE 4.4 COMPILE TIMES 72

TABLE 4.5 OBJECT FILE SIZES 74

TABLE 4.6 LINKER/BINARY STATISTICS 77

TABLE 4.7 OVERALL COMPILE TIMES 79

TABLE 4.8 OVERALL COMPILE TIMES COLUMN DESCRIPTIONS 81

TABLE 4.9 COMPILER DRIVER STATISTICS 83

List of Figures

FIGURE 2.1 CC/ACC/CFRONT: PREPROCESSOR/TRANSLATOR/COMPILER 12

FIGURE 2.2 CBACK: PREPROCESSOR/TRANSLATOR/REWRITER/COMPILER 13

FIGURE 2.3 OBJECTCENTER: PREPROCESSOR/TRANSLATOR/COMPILER/REPOSITORY 15

FIGURE 2.4 ENERGIZE: DATABASE/EDITOR/BROWSER/COMPILER 18

FIGURE 3.1 MINCPP: PREPROCESSOR/MINCPP/TRACE FILE/MINIMIZED OUTPUT FILE 22

FIGURE 3.2 MINCPP SYMBOL TABLE 24

FIGURE 3.3 TEXT CHAIN EXAMPLE 27

FIGURE 3.4 SIMPLE DECLARATIONS SYMBOL TABLE 34

FIGURE 4.1 EXAMPLES DIRECTORY STRUCTURE 50

FIGURE 4.2 WC.C INCLUDE TREE 52

FIGURE 4.3 TRIP_CB.C INCLUDE TREE 54

FIGURE 4.4 FONTS.C INCLUDE TREE 57

FIGURE 4.5 FILE SIZES: SMALL CASES 67

FIGURE 4.6 FILE SIZES: LARGE CASES 68

FIGURE 4.7 CPP/MINCPP TIMINGS GRAPH 70

FIGURE 4.8 COMPILE TIMES GRAPH 73

FIGURE 4.9 OBJECT FILE SIZES 75

FIGURE 4.10 LINKER/DEBUGGER GRAPH 78

FIGURE 4.11 OVERALL COMPILE TIMES: LARGE CASES 80

FIGURE 4.12 COMPILER DRIVERS COMPILE TIME 84

FIGURE 5.1 MINCPP SYSTEM 88

Abstract

`Mincpp`: Efficient, Portable Compilation of C Programs

George Moberly

Oregon Graduate Institute of Science & Technology

Supervising Professor: Tim Sheard, Ph.D.

The mapping of C programs into files introduces inefficiencies in the build process. Three quantifiable inefficiencies are: compile times too long, too many builds triggered, and, ironically, too few builds triggered. `Mincpp` solves these problems by replacing the C preprocessor in the compiler driver. `Mincpp` uses a C parser to minimize its output to just that needed, with all spurious source code omitted. The result is faster compile times, semantically unnecessary builds short circuited, and "correct by construction" file dependency generation. The result is a new development tool that leverages developer productivity by speeding up the edit-compile-link-debug cycle. `Mincpp` is portable to any system and works with any C compiler.

1. INTRODUCTION/PROBLEM STATEMENT

C programs are stored as files. C textbooks encourage a modular programming style where a program is stored in several files, the files reflecting the logical layout of the application [Harbison91] [Johnson86a] [Kernighan88]. Although not strictly required, most designs make use of "header files" (`.h` files), which are files that contain declarations that are shared between separately compiled modules (`.c` files). In fact, the programmer who does not strictly follow this practice runs the risk of introducing (difficult to find) runtime errors if declarations are mismatched between separately compiled modules. In fact, using libraries (`.a` files) created by the librarian utility normally involves including the declarations (function declarations and/or data structure declarations) via a preprocessor directive[1] that expands to the contents of the referenced file during preprocessing of the `.c` file, and including the library in the final link step.

Note that the above discussion references the C programming language in particular, but the observations and results of this work are at least conceptually applicable to a wide variety of programming language environments such as C++ [Ellis90] [Lippman91] [Stroustrup91], Modula, or Ada. In fact, this work was originally motivated by observations of slow compile times for C++ programs that used many header files.

The ANSI C standard [ANSI89] defines the logical stages of processing a "translation unit" into machine readable output, such as a binary compiled object file (`.o` file), interpreted output (such as p-code in the Microsoft C/C++ compiler [Microsoft91]), or some other implementation-defined binary representation. Transformation of the C module into a translation unit is the job of preprocessing [Jaescke89]. Preprocessor directives are processed, with conditional code, text substitution, and header file inclusion

---

[1] `#include <file.h>`

occurring. The translation unit is then suitable for evaluation by a C language parser.[2] While there is no requirement that preprocessing occurs as a pass over the input distinct from the compiler, many implementations do so using the standard preprocessor.

1.1 Hello, World

A natural result of placing declarations into header files is that the groupings of definitions are sub-optimal for any particular translation unit. That is, a typical translation unit will use *some* but not *all* of the declarations in any particular file. For example, consider the classic "hello, world" program [Kernighan88].

```
/* hello.c */

#include <stdio.h>
main()
{
        printf("hello, world\n");
}
```

This program requires a declaration for the `printf()` function from the standard C library to type check its call in the program. This declaration is in the `stdio.h` header file.

Note that all the C library I/O functions and types are declared in `stdio.h`, not just `printf()`. File `hello.c` just uses, or references, the single function prototype for function `printf()`. Here is a piece of a representative `stdio.h`:

```
/* stdio.h fragment */

int fseek(FILE *, long, int);
long ftell(const FILE *);
char *gets(char *);
int getw(FILE *);
int pclose(FILE *);
FILE *popen(const char *, const char *);
int printf(const char *, ...);
int puts(const char *);
int putw(int, FILE *);
void rewind(FILE *);
```

---

[2]The source code files (`.c` and `.h` files) are not interpretable by the compiler until preprocessing has been performed. The translation unit is the top level production in the grammar. The translation unit is logically a file whose contents are the preprocessor output. Most compiler systems can output this file by using the `-E` switch.

```
int scanf(const char *, ...);
void setbuf(FILE *, char *);
int sscanf(const char *, const char *, ...);
int ungetc(int, FILE *);
```

Now, consider a typical first build of `hello.c` using standard UNIX tools. These are a C compiler driver that uses a separate preprocessor, `make` [Feldman86] for generating build scripts, the standard C library for the `printf()` definition, and the standard linker. `Make` decides which modules need to be re-compiled based on file time-stamps. If the object file time-stamp is older than any source file time-stamp that it depends on, the build script is triggered. On a first build the object file is "older" (it is "infinitely old" in that it does not exist yet) so the compiler is run. The preprocessor expands the concatenation of `stdio.h` and `hello.c` as output, the compiler translates the result into an object file, and the linker searches the standard C library for a definition of `printf()` and finds it (hypothetically) in library component `printf.o`. If other functions are defined in `printf.o`, they are included in the `hello` binary executable.[3] Now, any subsequent modification to `hello.c` or `stdio.h` will trigger a build, regardless of whether the change is "important" to `hello.c`. In fact the only change that could matter in `stdio.h` would be for the function signature (declaration, or prototype) of `printf()` to change. `Make`, which is file-granularity based, cannot know this. So, `make` can and does generate many unnecessary builds because of the semantic gap between how the program is stored on disk (as files in a file system) and how the C language sees things (as one or many separately processed translation units).

Another problem occurs if the `make` dependencies are not correct. If a file dependency is missing and a change is made to that file that affects, say, the layout or alignment of a structure, a build may not happen that should have. This is a particularly difficult run-time error to diagnose. Having `make` not build in this case is worse than the previous problem of building too much.

---

[3]Well-designed libraries do not waste space in client's binaries. Commonly, one function is placed in its own C file, so the library members contain one function each.

1.2 Nova

We see that the size of the translation unit for "hello, world" after preprocessing is much larger than before.[4] The designer of `stdio.h` chose to include all the I/O declarations in one file rather than some other choice. Another choice might have been to place a few of the most used functions (`printf()`, `fprintf()`, `sscanf()`, etc.) in `stdio.h` and less-used functions (such as `fflush()` or `vfprintf()`) elsewhere. The less used functions could be placed in a separate file, say `estdio.h` (extended standard I/O). This procedure is called header file partitioning.

This new choice would result in a smaller `stdio.h`, and the expansion introduced by the preprocessor for this example would be much less. This is true since the one function used is one of the common ones. This design tradeoff is a common one: balancing the desire for small, efficiently processed, logically related header files versus fewer, more global, but more expensive to process header files.

For any particular client of a chosen header file, the partitioning is optimal from a compilation efficiency point of view only if the client actually uses every declaration in the header file. Unused declarations just waste time.

An ideal preprocessor would output the following for the example:[5]

```
int printf(const char *, ...);
main()
{
printf("hello, world\n");
}
```

This is `hello.c` with the included file expanded to just the single function prototype needed to check the call in the function body. The standard preprocessor instead expands the included file to a much larger quantity of text, a fragment of which

---

[4]This is not a general statement. For the specific example under discussion, this is true. In general, the size may be much larger. This tends to be true if many or long header files are read by the preprocessor.

[5]Ideal, or "perfect", is defined in terms of compilation efficiency in the sense of not requiring any unnecessary processing.

was excerpted earlier. Even using the proposed `stdio.h/estdio.h` arrangement, there are still other commonly used functions in the new, lighter `stdio.h`, so the standard preprocessor still produces non-optimal output.

Sub-optimal header file partitioning is almost a certainty. A given header file generally has many clients. These clients use different declarations depending on their individual algorithmic requirements. The only way to effect optimal header file partitioning is to dynamically materialize the header file for each individual compilation. This is an operational requirement for an ideal preprocessor.

The quantity of text expansion introduced by the preprocessor is of interest to us in quantifying potential gains in compiler efficiency. This expansion parameter will be defined as Nova (number of bytes in the `.i` file / number of bytes in the `.c` file), where `.i` is the output of the preprocessor. For "hello, world," this is $651/59 \approx 11$.

Contrast this with the "minimal" `hello.i`, also referred to as a `.min.i` file,[6] or `hello.min.i`. Here, Nova is $67/59 \approx 1$.

Programs that use lots of header files tend to have high Nova. Header files can also be deeply and widely nested, so the cost of inclusion can vary greatly depending on the hierarchy of sub-inclusion.

We now compare compiler timings for `hello.c` before and after minimization. The following is the compile time using the standard preprocessor:

```
time cc -c -g hello.c
0.2u 0.7s 0:02 45% 0+224k 15+2io 38pf+0w
```

and now the compile time using the ideal, or minimizing, preprocessor:

```
time cc -c -g hello.c
0.1u 0.5s 0:00 101% 0+200k 0+1io 0pf+0w
```

The important field in the `time` output is the third field, 0:02 for unaltered input and 0:00 for minimized input. This is the elapsed time for the operation. Minimization

[6]The standard preprocessor output is referred to as the `.i` file, while the ideal/perfect/minimal preprocessor output is referred to as the `.min.i` file.

reduces the elapsed time from 2 to 0 (nearly) seconds. The ideal preprocessor output is also referred to as the minimized output.

While the savings appears modest, we will see that as the size of the module increases, it can result in considerably less time spent in compilation. Compilations in large applications can take many minutes each to complete. If there are many modules to build, the elapsed time for the entire build to complete can be hours.

1.3 Thesis Problems

So far, three problems have been cited:

1. Excessive compile times because of sub-optimal header file partitioning.
2. Too many builds because of file-granularity dependencies.
3. Too few builds because of file dependency leaks.

`Mincpp` provides solutions to these problems. `Mincpp` is a C preprocessor replacement that integrates easily into any compiler system. `Mincpp` performs the dynamic header file materialization required for it to operate as an ideal preprocessor.

1.3.1 High Compile Times

`Mincpp` parses `.i` files with a C parser and builds an in-memory chained hash table structure for it [Knuth73]. Chains that represent the program text and type dependencies are also maintained. Using this structure, the tool can output only the required declarations that were used for this translation. File `hello.min.i` as shown above is an example output. Since the compiler sees less input, compile times are improved.

1.3.2 Too Many Builds

`Mincpp` could look at text differences after minimization and compare against previously materialized versions of the `.min.i` file. The absence of differences implies that the re-compile is spurious and can be avoided. This situation can easily occur if a shared header file is changed and the specific change does not affect a particular module.

For example, if the declaration of `sscanf()` is changed in `stdio.h`, `hello.c` does not need to be re-compiled since it refers only to `printf()`.

1.3.3 Too Few Builds

`Mincpp` could generate "correct by construction" `#include` lists and "`file.c : file.h`" `makefile` dependency information as a by-product of every build. The preceding reads as "`file.c` depends on `file.h`". This expresses a dependency asserting that `file.h` is included in `file.c`.

Since these lists are re-created as a by-product of every compile, there are no opportunities for a dependency change to be lost. This eliminates a source of run-time errors that are difficult to find. For example, a structure member is added to a header file. If the file dependency is missing from just one C module that uses that structure, `make` will not re-compile that module. That module now refers to the structure with incorrect offsets generated from the previous version of the structure. This is a hard problem to find even with a good window-based debugger.

1.4 Chapter Summaries

The remainder of the thesis is organized as follows. Chapter 2 presents related work. While `mincpp` is a new contribution to the development tools arena, other work to make the build process more efficient has been carried out and brought to market by others, and the problem of burdensome compile times, especially for C++, has been noted in the literature. This chapter considers related work for not only C, but for C derivatives such as C++ and Objective C.

Chapter 3 examines a sample implementation of `mincpp` that performs the `.i` → `.min.i` transformation. The system and its algorithms are presented in detail. This chapter defines an approach to solving thesis problem one.

Chapter 4 presents the results of using `mincpp` on five selected benchmarks. Reduced compile times are demonstrated.

Chapter 5 presents ideas for extending `mincpp`, and ideas for other tools based around `mincpp`'s basic architecture. Additional benefits beyond the core problems

addressed in the body of the present work are discussed. Also discussed are extensions to `mincpp` that address thesis problems two and three.

We end with conclusions in Chapter 6.

2. RELATED WORK

This chapter discusses the differences and similarities between `mincpp` and the following.

1. Traditional `UNIX` compiler environment.
2. `Cback` (NewCode Technology, Inc.).
3. Centerline ObjectCenter [CenterLine93].
4. Lucid Energize [Lucid93a] [Lucid93b].
5. Microsoft C/C++ [Microsoft91].
6. Borland C++ [Borland93] [Holub93].
7. IBM Tokyo Research Compile Server [Onodera93].
8. NeXT Smart Preprocessor [Litman93].

2.1 Overview

In the UNIX compiler environment, four different systems examined are: traditional compilers, `cback`, ObjectCenter, and Energize. In the MS-DOS/Windows environment, two different approaches to compilation are taken by Microsoft C/C++ and Borland C++. We will also describe systems from IBM and NeXT.

A summary of approaches used in these tools follows:

Table 2.1 Compilation Technique Summary

Technique	Used By	Description
Dumb Preprocessing	Traditional	Header files are expanded to their entire contents.

Source Code Rewriting	`cback`	C source code produced by the `cfront` translator is rewritten to improve and prune the code of unnecessary and unreadable constructs.
Interpretation	ObjectCenter	Source code is interpreted rather than compiled. This allows interactive evaluation as new code is entered into the system.
Compiled Header Files	Lucid ObjectCenter Microsoft Borland NeXT	The compiler state is dumped into a binary file when the compiler parses header files. If nothing has changed, the compiler can load the binary file instead of re-parsing the header files.
Incremental Compilation	Lucid	The system evaluates differences between old and new versions of the source code and re-compiles only altered functions and data.
Incremental Linking	Lucid ObjectCenter	Object files produced by the compiler are re-used to produce an executable binary. To contrast, the traditional linker reconstructs the binary from scratch every time the program is linked.
Compilation Server	IBM	A long-lived compile server takes requests for compilation and re-uses header files that have already been read into the compiler's internal tables.
Smart Preprocessing	NeXT	Header files are evaluated lazily. Header file declarations are pulled into the module on demand.

There are other environments for other languages that address the core problem of edit/compile/link/debug cycle performance, which is a crucial issue in developer productivity in general. Some examples of these include environments for languages such as Lisp, Smalltalk, and Basic. We shall not discuss these systems since we focus on C/C++.

The systems under consideration support C as well as C derivatives. Of the systems described below, `cc` and `acc` are C language only, `cfront` and `cback` are C++-specific tools, ObjectCenter, Lucid, Microsoft, and Borland support both C and C++, IBM supports a proprietary language based upon C that is object-oriented, and NeXT supports C, C++, and Objective C.

Chapter 5 will explore extensions to `mincpp` that would make it suitable for C++ as well as C environments.

2.2 System Descriptions

2.2.1 Standard UNIX Compilers: `cc`, `acc` and `cfront`

Probably the most common UNIX programming environment uses a traditional file-based compiler, using a "Kernighan & Ritchie" C compiler (`cc`), an ANSI C compiler (`acc`), or `cfront` as a C++ front-end to one of the C compilers. All program storage is based exclusively on UNIX files. The most common arrangement is for preprocessing to occur in a separate pass distinct from compilation, and for the compiler to generate assembler code that a macro assembler turns into an object file. This allows the compiler to not have to deal with the vagaries of macro preprocessing or object file formats.

Preprocessing was an area where compilers varied widely in the past, as the original C specifications were rather vague about its role in compilation. The ANSI C standard has cleaned up this area considerably. Consequently, ANSI preprocessing is common across most environments at the present time.

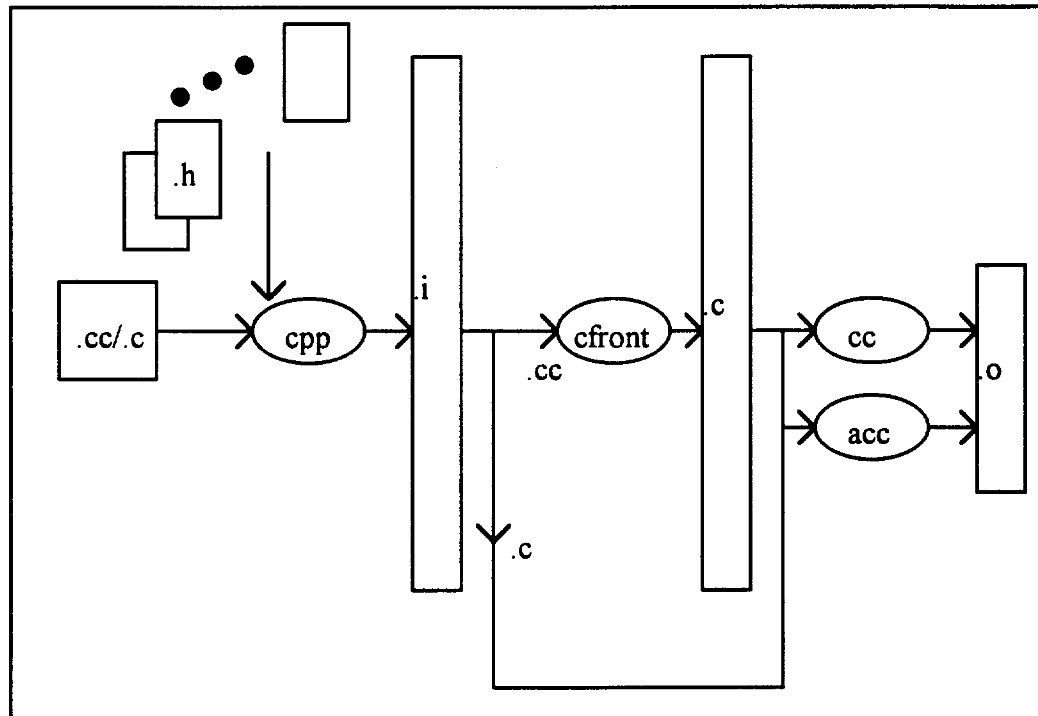

Figure 2.1 `cc/acc/cfront`: preprocessor/translator/compiler

Source code files are preprocessed. The result of this processing is either a C translation unit or a C++ translation unit. C++ translation units are converted to C translation units by `cfront`. C translation units are passed to the C compiler. The C compiler creates an object file.

Note that `cfront` is just an additional processing step over that already in place for a strictly C-based environment.

2.2.2 `Cfront/cback`

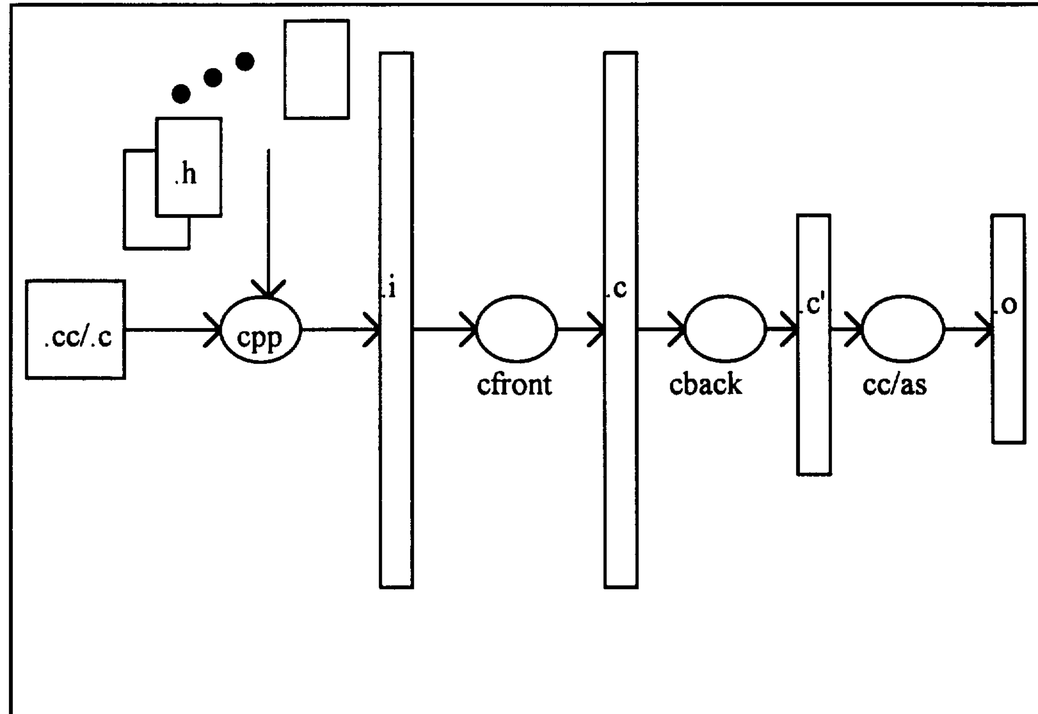

Figure 2.2 `cback`: preprocessor/translator/rewriter/compiler

`Cback` introduces another pass in the chain of processing, and is only relevant to `cfront`-based C++ compiler systems. The generated C code that `cfront` produces is very difficult to read. While today it is rarely the case that the programmer needs to look at this intermediate code, this was not always the case. Then, few debuggers understood "mangled names". Mangled names are the coded identifiers needed in C++ to differentiate between several overloaded functions and to refer to functions (or class methods) by their entire signature rather than just by name. The signature is an encoding of the return type of the function, the function's name, and an encoding of every parameter's type. In contrast, a function in C is uniquely identified by its name.

While `cback` was originally written to make the code easier to read, its later evolution to strip unnecessary type declarations from the C code is the feature relevant to this work. Type declarations that are "pure" type declarations can be stripped from the C code. A pure type declaration is one that does not declare space-allocating variables along with the type declaration. An example of a pure type declaration is:

```
enum boolean {True, False};
```

while a space-allocating declaration variant of this would be:

```
enum boolean {True, False} flag;
```

Other processing includes re-writing expressions, and eliminating unnecessary type conversions.

`Cback` reads the C code into memory and builds a parse tree. It also reads a file of rule descriptions that, essentially, direct ways of rewriting the tree that do not alter the semantics of the code, but result in the benefits described above.

`Cback` also reduces symbolic information in the object file by:

1. Eliminating unnecessary code
2. Eliminating unnecessary type/function/variable declarations
3. Simplifying expressions

This leads to reduced object file size, and reduces symbolic debug information. As a result, `cback` can reduce link time, and can considerably reduce debugger load time.

`Mincpp` shares some similarities with `cback`. Both read C code and "improve" it: `cback` targets improved readability, simplicity, and debugging issues while `mincpp` is primarily concerned with reducing compile time.

`Cback` and `mincpp` both parse input with a C parser, but `cback` builds a parse tree while `mincpp` builds a symbol table with dependency and text sequence links implemented as a chained hash table. `Cback` uses tree rewriting while `mincpp` marks reference information. Both output algorithmically-altered C code.

The majority of compile time is spent in preprocessing and translation. Because of this, `cback` cannot have a significant effect on compile times. Furthermore, the improvements that `cback` makes to the C code result in `cback` saving about as much time in the C compilation phase as it takes to run itself. While there are ancillary benefits to using the tool, it does not address the overall compile time problem that is the subject of the present work.

2.2.3 Centerline ObjectCenter

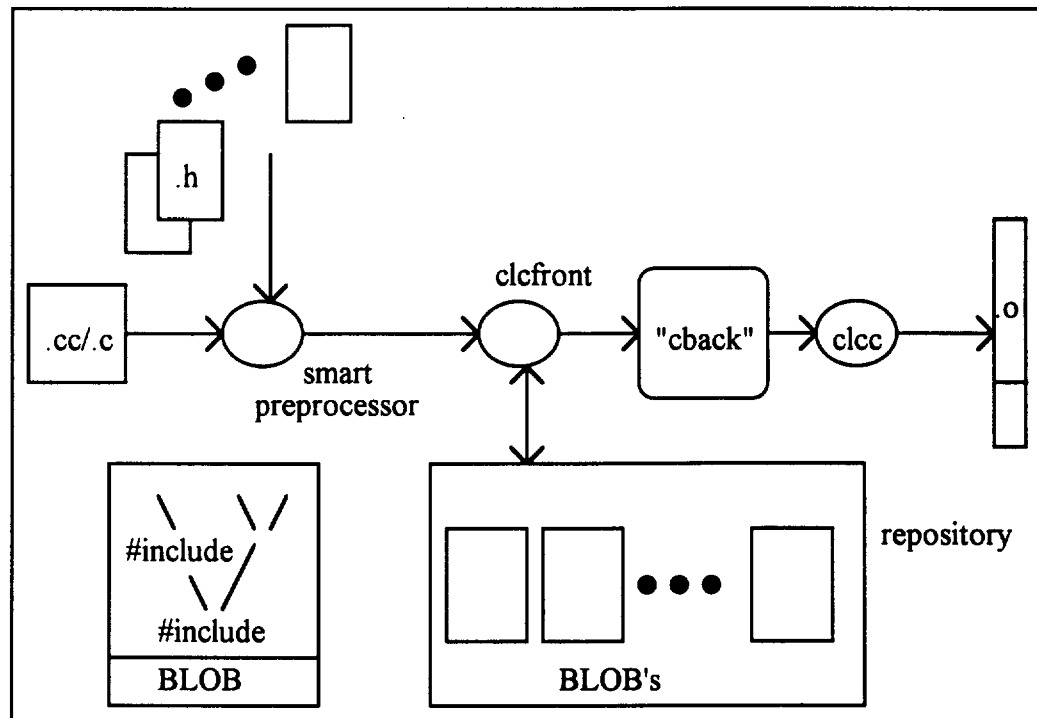

Figure 2.3 ObjectCenter: preprocessor/translator/compiler/repository

ObjectCenter is a complete programming system. It includes a graphical debugger, various language browsers, a program builder, incremental linker, integration with various source code control systems, a source code interpreter, and C and C++ compilers.

The interpreter allows interactive evaluation as code is being typed into the system. This provides good error checking and allows modules to be easily unit-tested by inserting stimuli and observing responses.[7]

Interpreted-code performance, both at load-time and run-time, is slow. One must be judicious in balancing interpreted and compiled modules. Typical practice is to use the interpreter only while doing initial development and testing, using the compiled module thereafter.

---

[7]A module can be tested by loading the module, and interactively calling functions to elicit behaviors. The calling of functions and examination of results can be driven by scripts and response files. This allows unit testing without a fully-built executable.

The compiler is a `cfront` derivative, and uses a modified version of the GNU preprocessor, `cccp`.

Most relevant to `mincpp` are two compiler features: compiled header files and "demand driven code generation."

Demand driven code generation does essentially what `cback` does, reducing unnecessary debug information by stripping unnecessary type declarations. Similar `.o` size reductions and debugger load times are observed. Compile times are slightly improved as less write I/O occurs when creating `.o` files.

Compiled header files are maintained in a "repository" directory. On an initial compile, `#include` preprocessor directives that are at the beginning of a module down to the first non-white-space token can be saved in the repository (this is an explicit restriction), and can later be restored instead of parsing. While the details of this process are proprietary, the repository entries are surely a binary representation of enough of the compiler's state to ensure that the compiler's state after loading a saved repository entry is the same as it would have been had it actually parsed the source. Some examples are in order:

```
// beginning of file
//
#include <stdio.h>
#include <string.h>

int i;

#include <malloc.h>

main() {}
```

In this example, the system will pre-compile the first two include files and stop when it sees a token other than white-space. There is no way to include the third file in the repository without altering the source code.

The pre-compilation of header files can be explicitly controlled by a preprocessor directive. For example:

```
// beginning of file
//
#include <stdio.h>
#pragma hdrstop
#include <string.h>
```

```
// and so forth as above.
```

Here the system stops the pre-compilation after the first include, as directed by the programmer.

Repository entries are tagged with the time-stamps of all their constituent files as well as the names and values of all preprocessor macros in effect for the entry. If macros or time-stamps change from compile to compile, the entry is invalidated, pre-compiled again, and saved.

As we will see later, the performance of this scheme results in only a modest improvement over just parsing the `.h` sources. Repository entries can be very costly in disk space usage if careful planning, restructuring, and rewriting of source code is not done.

ObjectCenter and `mincpp` have more differences than similarities. `Mincpp` is external to the compiler, and does not restrict the user's compiler choice. `Mincpp` defines a repository to pre-compile previous translations, but does so differently. This scheme is described in detail later.

ObjectCenter's compiled header files require the programmer to make source code changes to utilize the scheme effectively. Usually, this means creating a global header file and using it consistently throughout the application, and making sure that the compiled header is the first thing included in all the modules. The other systems that use compiled header files all document similar techniques. This artificial restriction does not work well with existing code.

There is also a temptation to create a single "global" header file that contains everything a program needs. Since there is likely to be a great deal of unnecessary information for any particular compilation, the savings from using the pre-compiled header are offset by the unnecessary information that needs to be processed.

2.2.4 Lucid Energize

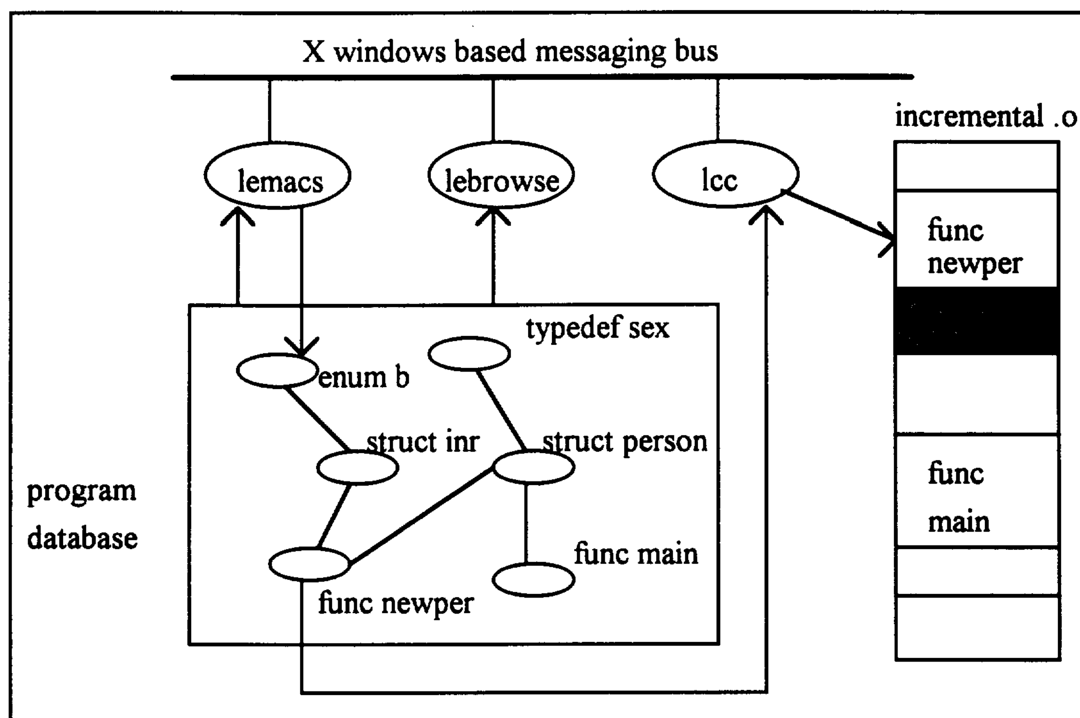

Figure 2.4 Energize: database/editor/browser/compiler

Like ObjectCenter, Lucid Energize is a full-featured programming environment. The feature relevant to the present work is an incremental compiler. The compiler can either run in standalone mode, producing a standard `.o` file, or in "Energize" mode.[8]

In Energize mode, the compiler breaks the source into "Language Elements", such as individual declarations, `typedefs`, and function bodies. It stores these language elements in an Object Oriented Database as individual objects. It also stores dependencies between the language elements in the database.[9] It also writes object code to a `.o` file that has "slots" for functions. These slots have extra empty "slop" space allocated in them. This allows new, larger versions of a function to be overlaid without recreating the entire object file. These `.o`'s are incrementally linked.

---

[8]Lucid's compiler also supports compiled header files. These are only available in the non-incremental version of the compiler. The operation of compiled header files is similar to ObjectCenter.

[9] For example, in the program fragment "`typedef int Boolean; Boolean T_F;`" The declaration of `T_F` is *dependent* on the declaration of `Boolean`.

To edit source code, either a conventional editor, such as `vi` or `emacs`, or "Lucid `emacs`" is used. If a conventional editor is used, the system evaluates text differences between the new and old versions of the file, and turns this into a list of altered language elements, which are marked as changed in the database. If Lucid `emacs` is used, the editor marks language elements as changed as characters are entered in the editor. Lucid `emacs` also graphically highlights altered language elements as the user types.

When re-compilation is requested, the compiler walks the dependency graph from all changed language elements, re-compiles only the affected functions, and swaps the newly re-compiled functions into the slotted `.o` file.

The structure of language elements can be browsed with a graphical "Language Element Browser", including navigation across the dependency graph.

The most striking similarity between Energize and `mincpp` is the organization of code into blocks, and maintenance of dependencies. `Mincpp`'s "text fragment" corresponds to Lucid's language element. `Mincpp` builds its text fragment list and dependencies in-memory rather than in a database. `Mincpp` does its processing external to the compiler, whereas Lucid's processing is done as a part of compilation.

While the Energize concept is an appealing one, performance of the system is, currently, poor. The compiler is considerably slower in Energize mode than in traditional file mode.

There has been a lot of research and patent activity recently in the area of different approaches for incremental compilation. For more information the reader can consult [Iitsuka93], [McKeeman92], or [Smith93] as well as the Lucid manuals referred to earlier.

Finally, note that the compiler in Energize is a "native" C++ compiler. This term refers to compilers that read C++ and write an object file directly. This is in contrast to ObjectCenter and `cfront`, where C++ is read and C generated instead of an object file. ObjectCenter links the translator into the C compiler, so this intermediate file is not generated. Other examples of native compilers for UNIX include those developed by Sun, HP, IBM, and DEC. Experience with these compilers suggests that compile times are not significantly reduced just by eliminating one intermediate file.

2.2.5 Microsoft C/C++

Microsoft C/C++ features compiled header files that work similarly to ObjectCenter's. As with ObjectCenter, a certain coding style is required to achieve a full measure of benefit from the scheme. Again, header files are stored in a repository. A repository entry is characterized by a set of macro bindings and an ordered list of header files. Each entry is stored in a separate file.

2.2.6 Borland C++

Borland C++ also has compiled header files. These header files may not contain code (functions). Again, source code changes are required to use the system effectively, since the same restrictions that were described for ObjectCenter and Microsoft are also restrictions in the Borland environment. All repository entries are stored in a single (large) file.

Both Microsoft and Borland are native compilers not derived from `cfront`.

2.2.7 IBM Tokyo Research Compile Server

This project implemented a long-lived compile server that can read header files into its internal context and re-use them for subsequent compiles. While the author is able to show reductions in compile time, it is done only at the expense of significant restrictions on what may be placed in header files. Also, the implementation is tied to a particular environment, and is not suitable for re-hosting. The use of a new, uncommon C dialect is a disadvantage.

2.2.8 NeXT Smart Preprocessor

This system comes closest to `mincpp`. Header files are manually compiled.[10] The preprocessor, like `mincpp`, parses text. The parser, on finding an undeclared reference, searches any encountered pre-compiled headers for a definition, and it is tagged for

---

[10]The file is pre-compiled by the user typing a command into a shell. The automatic pre-compilation process described for ObjectCenter/Microsoft/Borland is not available.

inclusion in the output. The net result is output that is similar to mincpp's. However, the algorithms and system implementation are very different.

NeXT requires manual work to pre-compile the header file. Also, it is necessary to flag the compiled header with #import, rather than a #include directive. This is a source code change that requires a special preprocessor to interpret.

NeXT uses a demand driven algorithm for including header file contents. The system waits for an occurrence of an undefined identifier. When this occurs, the declaration is located in a pre-compiled header, then marked for inclusion. The new declaration is then recursively re-parsed.

The lazy evaluation algorithm provides an interesting contrast with mincpp's greedy algorithm. Mincpp takes the entire preprocessor output and minimizes it without special preprocessors or source code changes.

2.3 Discussion

Most of the current approaches emphasize putting optimizations in the compiler proper. This thesis emphasizes that by minimizing the input to the compiler in the preprocessing phase of translation via a special purpose tool, compile times can be reduced. By being external to the compiler, the tool can work with any compiler in any environment. Unlike compiled header files, incremental compilation, or interpretation, mincpp can be easily used in any environment without change to work habits, source code, or makefiles. Furthermore, semantically-unnecessary builds are avoided altogether, a unique feature.

3. MINCPP

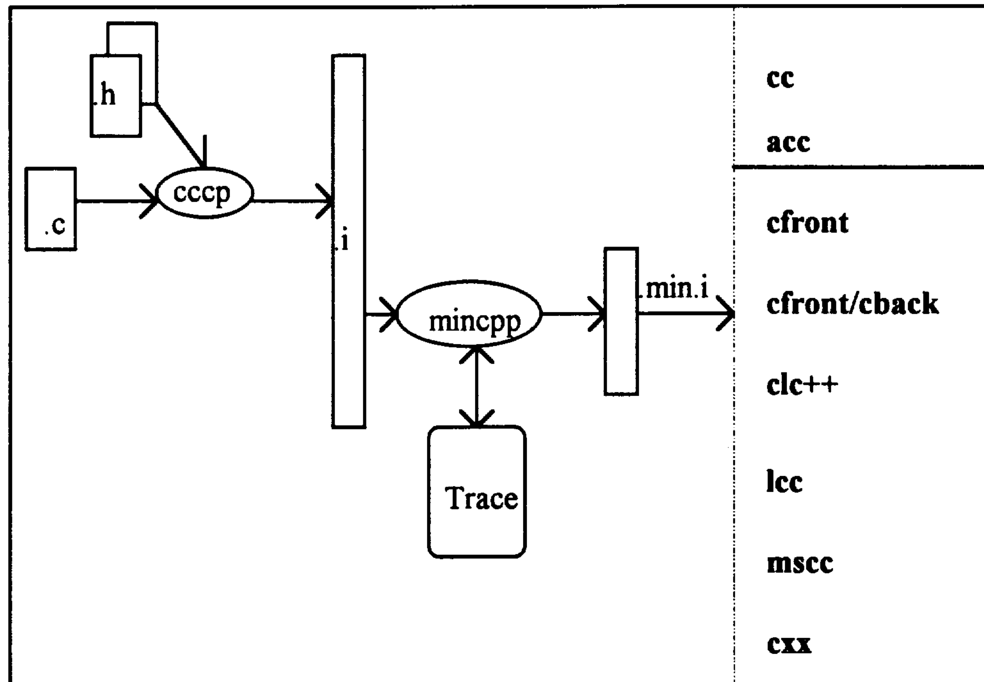

Figure 3.1 `mincpp`: preprocessor/mincpp/trace file/minimized output file

`Mincpp` reads input from either preprocessor output (the `.i` file) or from a trace file, which is an ASCII transcript of every symbol table manipulation routine called. `Mincpp` can be run once, reading the `.i` file: during this run a `Trace` file is created. `Mincpp` can then be run again, reading the previously created file. In either case, the output is a modified `.i` file that is suitable for passing to whatever compiler is used.

3.1 Implementation Rationale

`Mincpp` is a preprocessing phase after traditional preprocessing and before compilation. `Mincpp` minimizes its input and discards unnecessary declarations. As declarations are parsed, they are remembered. Later, when another declaration or code in

a function refers to the declaration, it is marked as needed. Unneeded declarations are not included in the final output.

One of the original goals of `mincpp` was to collect sufficient data to demonstrate compile time improvements. In doing so, several decisions were made to limit the scope of the project, so that the system could be implemented in reasonable time. The complete `mincpp` system that is described in Chapter 5 was originally planned. `Mincpp` and `cpp` were originally planned to be in the same binary. The parser was originally planned to parse C++ code as well.

By factoring out the I/O wasted in a separate pass, we can accurately predict run times, so the single binary work was deferred.[11] By parsing C syntax only, the work in constructing a grammar was greatly reduced.[12] The trace/replay cache is an either/or proposition: `mincpp` either parses 100% of it's input or it replays 100% of it's input.

Later, we will reason about cache effectiveness and the reader can draw their own conclusions regarding cache hit rates which one might expect to see. But, the replay mode allows us to quantify the potential benefits of using the trace/replay mechanism. Another potential approach would be to enhance the grammar specification to include preprocessing directives, and merge processing with `mincpp` into one pass over the input. This would cut additional time and boost compile time improvements.

For all its limitations, `mincpp` is still able to substantially improve compile times.

3.2 Symbol Table Basics

The key data structure in `mincpp` is the symbol table. The symbol table is hash table based [Aho86] [Sedgewick90]. The identifier is hashed into an index, and symbols that hash to the same index are grouped by a doubly linked list. Additional linking in symbol entries include a chain linking symbols of identical scope, dependency pointers,

---

[11]The joined, or single binary is the preprocessor code and the parser/minimization code put together in a single program.

[12]Originally, not much improvement on just C programs was expected, but the results on just C programs were suprisingly good.

and a program text chain. There are four types of links that are maintained in the symbol table. These are:

- Symbol link. Chains together symbols whose name hash to the same hash table bucket.
- Scope link. Chains together symbols whose scope is identical.
- Text link. Chains symbols together in the order their text appears in the translation unit. The full text of the translation unit is chained when processing ends.
- Dependency link. Chains symbols whose definition depends on another definition. An example of this is a declaration of a variable whose type is defined by a `typedef` declaration.

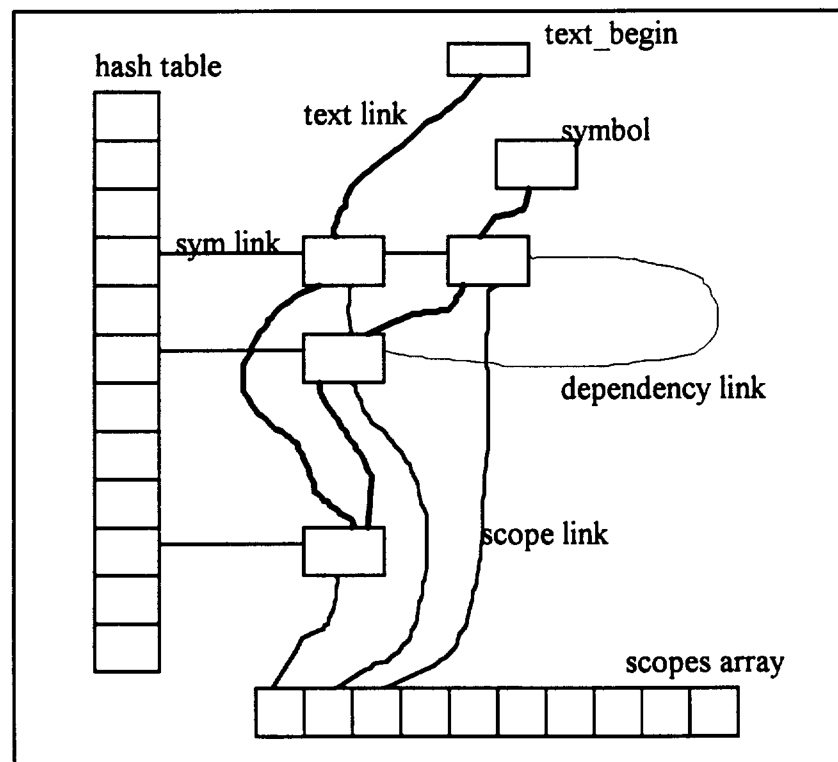

Figure 3.2 `mincpp` Symbol Table

The scopes array stores a chain of symbols of same scope. This speeds exit-scope processing as all symbols can be un-linked when they fall out of scope very simply. The symbol link is a standard symbol table technique. There is a pointer to the beginning of text. Each symbol has its program text associated with it, and the text fragments are

chained together in the order they appear in the source. These are shown as text links in the diagram. Finally, a dependency link shows a dependent relationship between two text fragments. For example, consider:

```
typedef int boolean;
boolean a;
int c;
main() {a = 1;}
```

This creates symbols named `boolean`, `a`, `c`, and `main` at scope 0 (file, or external scope) with text fragments "`typedef int boolean;`", "`boolean a;`", "`int c;`", and "`main() {a = 1;}`" respectively. There are text links between the four text fragments, plus scope links between the four symbols at scope 0, and a single dependency link between symbol `a` and symbol `boolean` to express the type hierarchy relationship between them.

On inspection, the minimal version of this program fragment is everything but `int c`. The reference to `a` in `main` marks "`boolean a`" as *referenced*, and "`typedef int boolean;`" is also referenced by the dependency link joining lines one and two of the text. References to a type have to also reference, recursively, dependent types. When the `mincpp` program exits, it traverses the text chain and prints out all the text fragments that have the reference bit set. Function definitions are referenced automatically. The symbols `a` and `boolean` are referenced by the mention of `a` in the function. The symbol `c` is never referenced, so it is not printed out.

Scope is incremented and decremented by seeing '`{`' and '`}`', respectively, in a function definition. Exiting scope also deletes that scope's symbols. There is a distinction between external declarations (scope 0) and local declarations (scope greater than 0). Local declarations do not have a text fragment, their text is part of the function definition's text fragment.

3.3 Scanner

The scanner, `lex.l`, turns the `.i` file into tokens for the parser, `yacc.y`. This is a very straightforward procedure, with just one subtle point. The scanner pushes identifiers and type names onto an "identifier stack" and "type stack". Identifier and

typedefname are terminal symbols in the `yacc` grammar. When the scanner recognizes an identifier, it queries the symbol table to see if it is a type via the symbol table's `isaType()` function. The scanner returns the character text of the token as its return type.

3.4 Parser Basics

`Mincpp` uses a complete C parser. The parser uses the matching text as the value passed through the `yacc` stack in all cases. Specifically `YYSTYPE` has type `char *`. `Yacc` puts the text matching the grammar production on the stack. This allows `mincpp` to easily grab text fragments at key points in the rule reduction sequence and index and chain the returned text into the symbol table.

The top level productions in the C grammar are:

```
translation_unit <= external_definition
external_definition <= declaration
external_definition <= function_definition
```

`Mincpp` breaks the text into blocks of external_definition granularity. Thus, an atomic text fragment is a single external declaration or external function definition. Text fragments are chained together as these productions cause text pointers to be updated in the symbol table.

As an external declaration is parsed (at file scope, or scope level 0, i.e. a global declaration), references to other declarations are noted and a dependency relationship is constructed by creating a dependency link in the symbol table. For example:

```
typedef int boolean;
struct s { boolean b};
```

Here the first declaration is independent, in that it uses only language-defined types (`int` is a *basic*, or built-in type). The second declaration is dependent on the first. If the second declaration is used, or referenced, this implies that the first declaration also is used. If, on the other hand only the first declaration is used, say the program uses `boolean` but never mentions `struct s`, then `struct s` is not used and the program will compile if it is not present in the text.

As function definitions, which include local declarations (declarations within function definitions, at scope > 0), and expressions (forming statements and statement lists, combining into compound statements) are being parsed, naming a type or calling a function (or taking its address) constructs a *reference* to the mentioned type, object, or function. For example:

```
int foo() {struct s s1;}
```

references `s` directly, and `boolean` indirectly, using the declarations given earlier, whereas:

```
int foo() {boolean b;}
```

references only `boolean`.

Consider the symbol table's text chain after processing the `boolean`-referencing function `foo`:

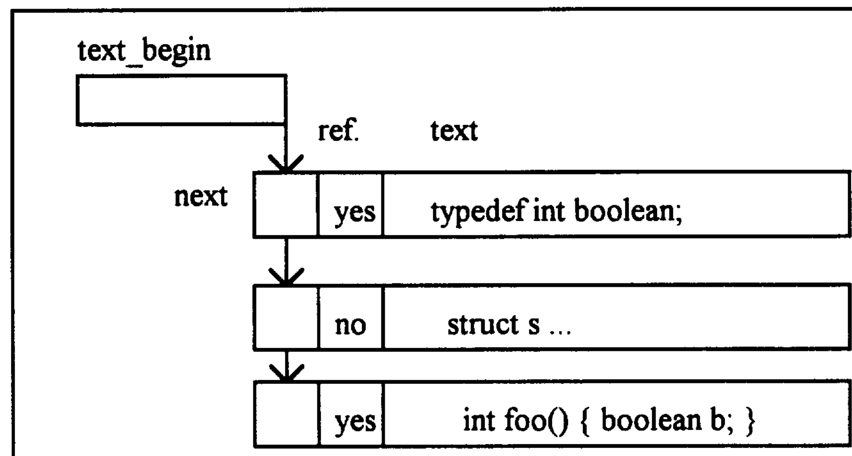

Figure 3.3 Text Chain Example

When end-of-file (`EOF`)occurs, the text chain is traversed, and only when *referenced* = *"yes"* is the text printed to standard output. This trims the text of external declarations that are not used somewhere. This example uses only pure type declarations, those that do not allocate space. Space-allocating declarations introduce additional considerations. These are elaborated further in the next section.

3.5 A Single Simple Declaration

We are now ready to give a detailed walk-through for a very simple program. `Mincpp` has three distinct tracing facilities that allow us to see in detail what happens during parsing, and how minimization is effected. Firstly, `mincpp` was written using Berkeley `yacc`. Berkeley `yacc` has a debug facility that outputs parse trees in a graphical presentation. This is an addition to the reduction trace available in standard `yacc`.

The `yacc` functions and `mincpp` contain code to trace their actions based on two conditional preprocessor definitions, `DEBUG` and `TRACE`. Lastly, the symbol table trace file that is used for replay gives a very detailed log of symbol table manipulations. Our strategy will be to present the source code fragment, then present the parse tree, debug, and trace output. Each will be accompanied by descriptive narrative.

Perhaps the simplest C program (even simpler than "hello, world") is

```
int i;
```

Here is the parse tree that Berkeley `yacc` produces:

```
          .... look ahead at INT   `int'
INT <-- `int'
basic_type_name
basic_type_specifier
|                  .... look ahead at IDENTIFIER   `i'
type_specifier
|       IDENTIFIER <-- `i'
|       paren_identifier_declarator
|       |                  .... look ahead at ';'   `;'
|       identifier_declarator
|       declarator
|       |       initializer_opt
+-------+-------+
|
declaring_list
|       ';' <-- `;'
+-------+
|
declaration
external_definition
translation_unit
|                  .... look ahead at end-of-file   `'
```

The key reductions are at *declarator* and *declaration*. In the output above, the text "look ahead" indicates where the parser has asked the scanner for another token.

Rules that end in _opt such as initializer_opt show places where the rule is optional. For instance, "`int i = 0;`" uses "= 0" as an initializer while "`int i;`" does not. `Mincpp` operates on text chunks at the external_defintion level of granularity. Here, there is only one such reduction.[13]

Next, the `DEBUG` and `TRACE` output:

```
/* symtab: add_symbol `NAMELESS`:0:object */
/* symtab: lookup i:-1  not found */
/* id_stack: pushed i */
/* symtab: isaTYPE IDENTIFIER */
type_specifier <- basic_type_specifier @i@
          r30c1 "int i"
/* type_stack: pushed BASIC */
declarator <- identifier_declarator @;@
          r78c1 "i;"
/* id_stack: popped i */
/* symtab: add_symbol i:0:object */
declaration <- declaring_list ';' @;@
          r26c1 "int i;" @  ";"
/* symtab: abort_tentative
symtab: add_snode int i;
/* type_stack: popped BASIC */
external_definition <- declaration @;@
          r76c2 "int i;"
```

Two classes of information are presented. The first format used is a pair of lines, the first containing a '<-', and the second indented. They show the rule reduced and the actual arguments to the function. Lines of the second format begin with the facility performing the transcription ("symtab" for the symbol table module, "id_stack" and "type_stack" for the stack managers) along with an operational description of the work performed. The presence of C-style comments was originally intended to allow this output to be included in minimized output. However, this was not consistently and completely implemented.

Referring to the DEBUG/TRACE output, the scanner looks up '`i`' and pushes it on the "identifier stack" because it is not a type name. When "`int`" is recognized as a type_specifier of "basic" type (`int`, `char`, `float`, `double`, `long`, `short`...), it pushes "BASIC" on the "type stack". The meaning of this is to inform a later action that the type of the declaration is language-defined and not dependent on anything else. The

[13]There is only one declaration in the program, so there is only one external declaration.

declarator function pops the identifier stack and creates a new symbol for the declarator. When the entire declaration is recognized, the type stack is popped, and the text of the declaration is associated with the symbol `i`.

And finally, the `Trace` file for this run:

```
2@`NAMELESS`@0@0@0
2@i@0@0@0
17@2
10
14@int i;@0
13@2
```

A line is printed to this file for every symbol table operation performed. The function called is encoded as a number in the first field. The arguments to the function are then given, all separated with a '@' character, which was chosen because it is not a legal character in a C program. This allows a compact representation to be easily parsed by successive calls to the C `strtok()` facility. As we will see in more detail later, this file is used by `mincpp` in "replay" mode.

At line 1, field 1, and line 2, field 1, '2' is the code for symbol table function `add_symbol()`, and field 2 is the name of the symbol being inserted. NAMELESS is a built in symbol whose meaning will be discussed in a later section. Symbol `i` is added next.

Symbols are numbered as they are inserted. They are numbered numerically in ascending order. In this example, symbol `NAMELESS` is symbol 1, and symbol `i` is symbol 2. These numbers are referenced as fields in the file later on to add additional information to the symbol.

At line 3, symbol `i` is tagged as needing text. As we will see, some symbols do not need text, such as local declaration symbols. At lines 5 and 6 the text "`int i;`" is associated with the symbol `i`. When `EOF` is recognized, the text chain is traversed. There is only the single node for symbol `i`, and it has not been referenced by anything else, so it is not printed in the minimized result.

`Mincpp`, as implemented, deletes un-referenced space-allocating declarations. This is not a sufficient condition for a general-purpose tool. See section 5.6.3 for a complete treatment of this subject.

3.6 Symbol Table: Part 2

We will present the symbol table data structure used by `mincpp` and discuss its parts:

```
typedef enum kind
{
  object,
  function,
  typedef_name,
  enum_constant,
  label,
  su_tag,
  enum_tag
} Kind;

typedef enum boolean
{
  True = 1,
  False = 0
} Boolean;

typedef struct symbol *Symbol;

struct symbol
{
  char *text;
  void *textp;
  Boolean ref;
  char *name;
  int scope;
  Kind kind;
  Symbol next;
  Symbol prev;
  Symbol slink;
  Symbol dep[16];
  long id;
};
```

The members `name`, `scope`, `kind`, `next`, and `prev` are members that are to be expected in any sort of compiler symbol table. They are minimally needed to track the symbol's name, lifetime, type, and links to the symbol table's hash structure. The `slink` member is an optimization that allows for quick exit-scope processing. The new members that facilitate minimization are `text`, `textp`, `ref`, and `dep`. The remaining member, `id`, is introduced to allow symbols to be persistent. This is needed for the cache mechanism described later in this chapter.

A member-by-member description follows:

`Kind` tells what class of symbol an instance is. These classes are taken directly from K&R 2e [Kernighan88]. There are different name-spaces associated with these classes, and this tag allows name-space discrimination to be accomplished on lookup and insert. Since only one symbol table is kept instead of one per name-space, this is an important thing to keep track of. This feature is also critical in discriminating between type names and objects which is required for the scanner to correctly return terminal symbols identifier and typedef_name to the parser.

Member `boolean` is used to mark symbols as referenced. Clearly, without a marker there is no way to keep track of unused code.

Member `text` is a pointer to the program text that corresponds to the external_definition a symbol belongs to. `Textp` is a pointer to the next symbol in lexical order, or text chain order. `Name` is the name of the symbol and is what is matched on lookups. All symbols of same scope are linked by `slink`, which allows symbols to be deleted when they fall out of scope by just chasing the `slink` pointers. `Scope` tells the lexical scope of the symbol. This field is actually redundant, because of the `slink` member. An earlier implementation of the symbol table did not have this, and the table had to be searched on the `scope` member to delete out of scope symbols. `Next` and `prev` are forward and backward doubly linked list pointers that link symbols that hash to the same bucket.

Member `dep` is an array of symbol pointers to symbols upon whom the symbol is dependent. This is a poor implementation since most symbols have fewer than this, and the processor breaks if there are more. This should be a dynamically allocated list of pointers. The present implementation is very memory-inefficient but was done for expediency.

Finally, `id` is a unique tag for a symbol that allows symbol identity to be established in the persistent trace file. We will see later how this is used in more detail..

3.7 External Declarations

`Mincpp` organizes a program as a list of external definitions. These are either scope 0 declarations, or function definitions. All function definitions are marked as

referenced when encountered. Declarations are marked as referenced as the expressions and declarations in the program make use of them.

We will cover each category of external declaration and present an example.

3.7.1 Simple Declarations

Simple declarations are declarations that use only built-in types. They can have declarators that denote pointer or array variants. For example:

```
1    int i, j;
2
3    char *c, c1[128];
4
5    int foo()
6    {
7      c = 0;
8    }
```

This program has three text fragments, one for the `int` declaration, one for the `char` declaration, and one for the function definition. Only `c` is referenced. These declarations each have two *declarators*. Declaration one has declarators `i` and `j`, declaration two has `*c` and `c1[128]`. `Mincpp` inserts symbols, at scope 0, for declarators `i` and `j`. When the declaration is recognized, it attaches the text "`int i,j;`" to both symbols, and links the two symbols into the text chain. Then, symbols `c` and `c1` are inserted into the symbol table at scope 0, the text `char *c, c1[128];` is attached to both symbols, and the two symbols are linked into the text chain.

When `foo` is recognized as a function definition, it is inserted as a function symbol at scope 0 and its `ref` member is marked `True`. When "`c`" is seen as an expression component, `mincpp` looks up `c` and references it. Visually, the processing can be shown as:

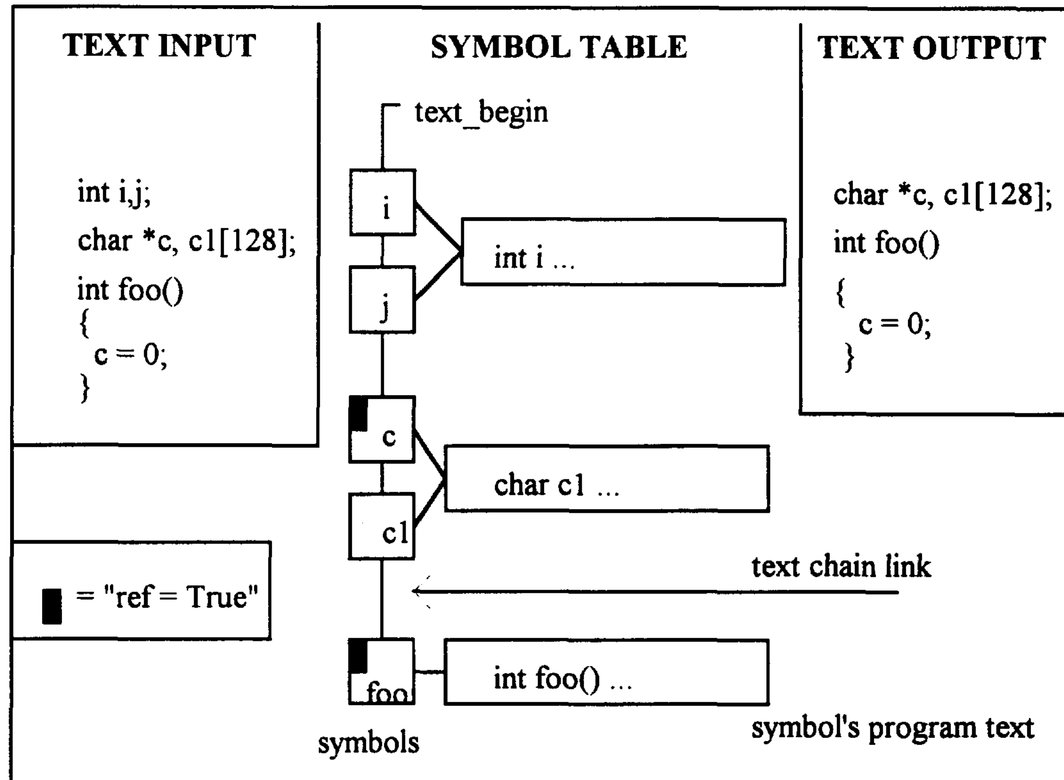

Figure 3.4 Simple Declarations Symbol Table

Upon `EOF`, `mincpp` visits the text chain starting with `text_begin`, and prints every text fragment for symbols that have *ref* = *True*. In this case this is true for symbols `c` and `foo` only. In answer to the question "what if both `c` and `c1` are referenced", `mincpp` stores a pointer that keeps track of the last text fragment printed. If the current text pointer equals the previous pointer, the text is not printed twice.

The granularity of text disclosure is the external declaration. So, in this example, `c` and `c1` both map to the same external declaration, so it is disclosed if either is referenced.

In this example, if either `i` or `j` is referenced from another module through an `extern` declaration, it is wrong to not disclose them here. While many programs, such as the benchmark programs described later, do not use global variables in this fashion, it nevertheless is wrong to assume this. The version of `mincpp` described here *does* perform these un-safe minimizations. Section 5.6.3 describes enhancements to `mincpp` that *do* allow such declarations to be safely minimized. `Mincpp` can be altered to not perform these minimizations by the addition of a small amount of logic. When the

declarator symbols are inserted, a flag is set. Later, when the text is entered, the flag is checked. If set, the symbols are marked as referenced. This allows minimization of pure declarations while keeping space-allocating declarations. This behavior should be the default behavior for the tool as described here.

Declarations with multiple declarators could be further broken up into individual external declarations. We could re-write a declaration such as:

```
int x, *y, z[3];
```

as:

```
int x;
int *y;
int z[3];
```

This approach was not taken. The result from this example is that if *any* of `x`, `y`, or `z` is referenced, then all three are declared.[14]

3.7.2 Function Prototypes

Function declarations are one of the more interesting "dark corners" of C. In K&R C, functions did not even have to be declared. They were assumed to return `int` unless something else was specified. Only in ANSI C [ANSI89] were *function prototypes* introduced, which allowed the number and type of function arguments to be placed in a header file so the compiler could check types, previously a notorious source of run-time errors. ANSI C grandfathered the K&R rules and syntax, but additionally allowed function definitions to use prototype syntax. Another interesting ANSI addition was the `void` type. Because of the K&R syntax, not specifying function parameters in a prototype really meant that the type and number were unspecified. To say that a function actually took *no* parameters required saying `function(void)`. Just to spice things up, C++ demands a function prototype and excludes the K&R defaulting syntax, which means that `function()` in C++ *does* mean function taking no parameters. It should be noted that

---

[14]At one point the implementation *did* break these up. While some data space savings might occur, the implementation complexity was deemed not worth the effort.

some of these ideas were originally created for C++ and later wound up being accepted into C by the ANSI committee. A few examples should prove useful:

```
/* K & R syntax */

func();     /* function declaration                                  */

func(i, j) /* function definition, matches either explicit decl.    */
           /*   or will default if missing                          */
  int i;
  float j;
{ ...}

/* ANSI C accepts all of the above, and introduces the following */

void func(int, int j);    /* function declaration prototype,          */
                          /* only type names                          */
                          /* are required, variable names can be      */
                          /* added for documentation purposes         */

void func(int i, int j) { ... }
               /* function definition using prototype argument syntax */
```

`Mincpp`'s parser accepts any combination of the above-mentioned syntax. Our example program follows:

```
1   int foo();
2   int foo1(int a);
3   int foo2() {foo();}
```

Function parameters present an interesting problem for `mincpp`. When a function parameter is recognized, we do not yet know if we are in a function prototype declaration or in a parameter section of a function definition. Also, ANSI requires the scope of function parameters to be the same as the scope of the symbols immediately after the "{" that starts the function body. In other words, function parameters are conceptually inserted as symbols right after the "{", or right before the first local declarations. Also, in a prototype, the parameter name is optional. All that is needed is the type, called an **abstract_declarator** in the grammar.

`Mincpp's` approach to this is to add a *tentative* concept to the symbol table facility. When a parameter name is encountered, either in a prototype or parameter section, it is inserted into the symbol table tentatively. The symbol table facility stores tentative symbols in a separate place distinct from the hashed structure. Later, in the case

of a prototype with named arguments, function `abort_tentative()` is called to delete these. In the case of a function definition, after the initial "{" is seen, function `insert_tentative()` is called to make these symbols first class symbols in the hash structure.

In our example, `foo` is recognized as a function when the "()" parses to a postfixing_abstract_declarator. It is inserted as a function in the symbol table, and its text is associated when the declaration is complete. `Abort_tentative()` is called with no effect since no parameter names were present. At line 2, symbol `a` is first added tentatively when it parses to a parameter_declaration. This occurs before the function is parsed to a postfixing_abstract_declarator, at which point `foo1` is added as a function symbol. When a declaration is recognized, text is associated, and `abort_tentative()` has the effect of erasing the tentative "a". As before, `foo2` is added and referenced upon encountering the "()". When "foo" appears in a function call expression, `foo` is looked up, found at scope 0, and referenced. At `EOF` the program, without the declaration of `foo1()`, is printed to standard output.

3.7.3 Enumerations

Enumerations are the first construct that introduce *tags* and *members*.[15] Enumeration tags can be used to refer to an `enum` declaration later in the program. The members are just integer-valued identifiers that allow the intent of an assignment in the programmer's mind to be made self-documenting. Our example program follows:

```
1   enum {red, yellow, green} a;
2   int foo() {a = red;}
3   enum boolean {true, false};
4   enum boolean b;
```

At line 1 we see an example of a *tagless* enumeration. It declares three members that might represent stop light colors. Here, `a` is a variable declaration. The variable can be assigned the enumeration constants, as the members are sometimes called.

---

[15]In the fragment "`enum temperature {cold, warm, hot};`", `temperature` is the tag, and `cold/warm/hot` are the members.

Mincpp sees the constants reduce to non-terminal enumerator_list. When this happens, the constants are placed into the symbol table in their enclosing scope, here scope 0. Now, a is recognized as a declarator once more, but, for the first time in our examples, this type is non-BASIC. Nevertheless, a is inserted as an object, or variable symbol. When declaration is reduced, all four symbols are tied to the text of the declaration.

In foo, both a" and red are referenced in two expressions that constitute an assignment statement, which then reduces to a { }-delimited compound statement. a and red are looked up and referenced in turn.

At line three we see an enum with a tag. When parsed, "boolean" is inserted as an enumeration tag. The constants "true" and "false" are inserted as before, and the three symbols are linked to the declaration text.

Lastly, "boolean" is pushed on the type stack, and b is a declarator. The declarator code sees "boolean" on the type stack. This is a *dependency*, not a *reference*. The declarator code adds boolean as a dependent type. The program terminates with EOF. Now, enum {red, yellow, green} a; and the function are printed. Since neither boolean or b were referenced by an expression, neither are printed.

3.7.4 Structures and Unions

The most complicated declarations in C are those for multi-membered data structures, or struct/union types. The basic syntax is (struct | union) optional_tag '{' member_list '}' optional_identifier_list ';'. If the identifier list is absent, no variables, or objects that set aside storage, are declared, and this is a true type declaration. If the tag is not present, then no variable declaration that occurs lexically later in the program can refer to the structure. Members can be any storage-allocating declaration, including other structures. Structures can be self-referencing.

Mincpp has a concept of the "current" external struct/union symbol. It uses this to add dependencies to the current external declaration being parsed. The strategy is to collapse all the members under the external struct into dependency links and ignore the details of the members. An example will be helpful.

```
1   struct a {int i;};
2   struct b {struct a aa; float f;};
3   int foo() {struct b bb;}
4
```

Identifier "a" is placed in the symbol table as a struct/union tag. Since we are at scope 0, symbol "a" is designated to be the current external structure/union. The grammar parses the members, (the ';' separated list of declarations inside the struct brackets), as before. The key difference is that the declarations reduce to non-terminal member_declaration rather than declaration, and member_declaration loops via member_declaration_list. As we saw before, "int i" pushes "BASIC" on the type stack. When we see the declarator "i", we check to see if we are in a struct, and since we are, "i" is *not* placed in the symbol table. This time, we reduce to member_declaration. Instead of the process of linking text to declarators as before, we just look at the type stack to see if the type is BASIC or not. This time it's basic so there is nothing more to do. Now, the declaration reduces, and we add the text of the declaration to symbol "a".

For the second declaration, (that of "b"), we insert "b" in the symbol table as a structure/union tag. The first member pushes type "struct a" on the type stack. When the member_declaration is reduced, we pop the type stack and see "a", not "BASIC". We lookup "a" and add a dependency link from "b" to "a". The rest of the processing is similar to that described for "a". Here, we just show one dependent type, but there can be an arbitrary number of types that attach themselves to "b".

The function foo has one local variable, bb. The local declaration is parsed just like an external declaration, with one important difference. Local declarations that refer to non-BASIC types cause a *reference* to occur instead of a *dependency*, as is the case for scope 0 declarations. There are some other issues regarding local declaration that will be deferred until later in the section devoted just to them. The key point is that the reference to type b in the local declaration marks the symbol b as referenced. As the dep array in symbol b has a pointer to a, we follow the pointer and also mark a as referenced. This

algorithm does a depth-first tree walk of the dependency arrays to mark as referenced the type dependencies.

It is now time to explain the *NAMELESS* symbol that always is placed in the symbol table at program startup. Recall from earlier discussion the fact that the tag of a `struct` declaration is optional. This allows declarations such as

```
struct {int x; int y;} Point;
```

But now consider the following program:

```
typedef int temp;
struct {int i; temp t;} a;
```

Ignoring the specifics of `typedef`, which is the subject of the next section, the notable thing here is the lack of a tag *and* a type dependency between the variable `a` and the `typedef temp`. For such instances, a special variable, *NAMELESS*, is made the current external `struct`. Now, a dependency between *NAMELESS* and `temp` is made for the second `struct` member. Later, when the whole declaration is parsed, we copy the dependencies from *NAMELESS* to "`a`", which is the only symbol table symbol in this example. Had we said "`a, a1, a2`" instead of "`a`", the dependencies would have been copied to each in turn. After copying, *NAMELESS* dependencies are cleared to ready it for its next use.[16]

3.7.5 Typedef Declarations

`Typedef` in C does nothing more than provide a handy way to provide an alternative name for an existing type. The existing type can be built-in or user specified (`enum/struct/union`). A simple example to explain `mincpp`'s treatment of `typedef` names should suffice:

---

[16]*NAMELESS* is a poor choice of name for this symbol as it could collide with a program name. A better name would be ``` or `$`, characters other than @ which are not part of C's character set. The character @ cannot be used as it is already reserved to delimit fields in the trace file. Also, the special symbol for this purpose could be kept outside of the symbol table, but keeping it inside allows for a simpler and cleaner implementation of tagless structures.

```
1   typedef int temperature;
2
3   temperature t;
4
5   int foo() {t = 100;}
6
```

The keyword `typedef` has the effect of putting `mincpp` into a mode where declarations do not populate the symbol table as objects but instead adds a type name symbol. The declarator is inserted into the symbol table as a typedef name rather than as an object. The end-of-declaration processing ties the text to the typedef name symbol `temperature`. At line 3 the type_specifier code again pushes `temperature` on the type stack. The declarator code creates a new object symbol for `t`, and the declaration code pops the type stack, creates a *dependency* from `t` to `temperature`, and links the text. Inside the function, `t` reduces to a primary_expression, which causes scope 0 `t` to be referenced, as well as `temperature` via the dependency link.

3.8 Function Definitions

So far, we have been concerned only with external declarations. The other syntactic element at external_definition level is the function definition, or body. As we saw earlier, `mincpp` treats the entire block of a function as a single text unit. For function definitions, the overall strategy is to alter the behavior of local declarations to cause them to be in *reference mode* rather than *dependency mode*. We also create local declaration symbols for reference but do not tie any text to them, so they are not put in the text chain, and we delete them when they fall out of scope. Lastly, function parameters require some special treatment, especially the ANSI prototype form.

Nothing inside a function is minimized. They are parsed only to see what they reference.

3.8.1 Old-Style Parameters

A function parameter can be thought of as a local variable in scope 1 that is logically inserted into the symbol table right after the '`{`' that opens a function body.

```
1   int a;
```

```
2   int foo(a)
3   int a;
4   {
5     a = 100;
6   }
7
```

Here, scope 1 a is what the expression at line 5 involving a references, not scope 0 a, which has been hidden by the local symbol with the same name. The grammar has a non-terminal, old_function_declarator, that is used when the function parameter list is closed by the end parenthesis and the next token is *not* the open brace that starts the body. This sets a flag that indicates that we are parsing declarations that are function parameters. So, when declarator "a" is parsed, it is inserted at *scope + 1*, or just scope 1 in this example. At declaration end, we see that this flag is set and do not tie any text into the text chain, since elements in the text chain are just those of external_definition granularity. The expression involving a causes a to be looked up, and the most deeply nested one is returned, or the scope 1 a here. When we fall out of scope 1 at the end brace, all scope 1 symbols are deleted, or just scope 1 a that has its reference field set. So, scope 1 a came and went, and the only real effect was to cloak and hide the external a so it would not get its reference bit set. Finally, once again we tie the function text into the text chain when the external_definition for function_definition is reduced.

3.8.2 ANSI-Style Parameters

A variation on the parameter theme is function definitions that use the ANSI prototype syntax.

```
float f;
int foo(float f)
{ f = 2.0; }
```

Scope 0 f is processed as usual. Inside the parenthesis, f reduces to parameter_declaration.[17] This causes f to be inserted *tentatively*, meaning it is stored

---

[17]In the treatment of function declarations, we saw that the identifier could be missing. This is known as an abstract parameter in that the type but not name is declared. C++ extends this syntax to the function definition parameters as well. This allows the programmer to explicitly state that this is a

away internally by the symbol table but not really put in the real hash structure. This is because we do not yet know if we are parsing a function declaration or a function definition. We have seen this before when discussing function declarations. The open brace in this case triggers a utility function called `function_def()`. One of the things it does is to call the symbol table `commit_tentative()` routine. This takes the tentative list and promotes them to be real symbols. It inserts them at *scope + 1* since the helper non-terminal that increments scope on recognition of the open brace has not yet reduced (but it will soon). So, here `a` is promoted to scope 1 and the body parsing begins. Just as for an old-style parameter, `f` references scope 1 `f`, then scope 1 falls out of scope after having hid scope 0 `f` from being referenced, scope 1 is exited upon '`}`', and scope 1 `f` is deleted.

3.8.3 Local Declarations

Function parameters, which we have already treated, are just a special case of local declarations. They are inserted and deleted as they fall in and out of scope. The trick to insert the parameters at *scope + 1* is not needed.

```
1    int a;
2    int foo()
3    {
4      int a;
5      a = 7;
6    }
7
```

The only difference between this and the old style example is that the local variable is a true scope 1 variable rather that a parameter scope 1 variable.[18] Scope increases as more open braces are seen, indicating statement block begin and scope decreases as close brace is seen, indicating statement block end. The final close brace terminates the function body, and all scope 1 locals, either "real" or "parameter" are deleted. Scope > 1 variables are deleted as the '}' that closes the scope is encountered.

---

parameter to the function, but that it will not be used. This can allow control over compiler warnings about unused parameters.

[18]This can chain indefinitely. Consider `int foo() {int a; {int a; a = 9;}}`

3.8.4 Referencing

This concludes our discussion of specific language constructs that we will examine. Before proceeding we will consider one more example:

```
1    int a;
2    int b;
3    char **c;
4    struct s {int i;};
5
6    int f1()
7    {
8      struct s ss;
9    }
10
11   int f2()
12   {
13     c = 0;
14   }
15
```

Variables `a`, `b`, `c`, and `s` have symbols created and are then chained to the text, since they are external declarations. Function `f1` is placed in the symbol table, and references `s` through the declaration for `ss`. `C` is referenced by `f2`. At the end, `a` and `b` are not printed since they are not referenced, `c` and `s` are printed since they are referenced, and the two functions are printed since function bodies are always referenced.

3.9 Save and Replay

When the development of `mincpp`'s parse mode was complete, an investigation into strategies for boosting performance was conducted. Profiling the application [Pure93b] revealed that 96% of the program's execution time was spent in `yylex()` and `yyparse()`, in other words, the majority of processing time was spent parsing. This was a welcome result, as it showed that the symbol table and other processing that `mincpp` does did not significantly add to the time just to recognize input as C code via parsing. It also suggested that an approach for saving `mincpp`'s state in an external file and replaying it later would result in reduced overall processing time.

3.9.1 Save

On an initial pass, the trace for the entire translation unit is saved. On a subsequent run, the trace can be replayed instead of parsing the same translation unit. This is ineffective as an actual caching mechanism since it does not accommodate any changes in the files, but it is useful for setting an upper limit on cache efficiencies. Fully parsing gives a worst case, or zero percent caching limit. Replaying gives a best case, or 100 percent caching limit. The effectiveness of caching is dependent on the nature of change to the files. Local changes to C modules and header files will re-use most of the trace information and caching will be effective.

All of `mincpp`'s state is encapsulated in its symbol table. Any manipulation of the data structure is done through the symbol table manipulation functions prototyped in `symbol.h`. There is a clean interface that resembles a C++ class with all data members private. The code is, however, just C code that obeys abstract type coding rules.

Save works by adding a single field to the symbol structure, "`id`", which contains a unique symbol number. This is done because the in-memory symbol address is not persistent and is only valid for the current execution of the program, and we need a way of identifying symbols persistently.

Every symbol table manipulation function had a line of code added to it to append a line to a file (the `Trace` file) giving the operation performed and the function arguments. Each function was assigned a number, or operation code. It writes that instead of the name of the function, mostly to save space. Most of the symbol table functions take symbol pointers as arguments. A `Dictionary` class was added to store associations of "symbol id ↔ symbol address" pairs. This allows the functions to lookup the `id` for a symbol address and transcript the `id` to the `Trace` file instead of addresses. The dictionary gets new pairs from the `add_symbol()` routine.

3.9.2 Replay

On startup, the environment variable USE_TRACE_CACHE is examined. If it is set, we replay by calling `load_sym()`, otherwise it parses by calling `yyparse()`.

Load_sym() opens and reads the trace file one line at a time. It dispatches on the operation code it finds and calls the specified symbol table function. If the operation is to create a new symbol with add_symbol(), it reads back the memory address of the newly created symbol (add_symbol() returns this) and creates a dictionary entry for it. If the operation has a symbol address as an argument, it translates the symbol id from the on-disk file into the memory address by looking up the id in the dictionary.

The Dictionary class takes addresses and turns them into id's on behalf of the manipulation routine's need to create the file, and takes id's and turns them into memory addresses on behalf of load_sym()'s need to have memory addresses to call the manipulation functions.

3.10 Using Mincpp

One approach to using mincpp is to change the make rule used. In GNU make [GNU92], mincpp can be used by changing a pattern rule like:

```
%.o : %.c
        cc -c -g $<
to:

%.o : %.c
        cccp $< > tmp.i
        mincpp < tmp.i > tmp.min.i
        cc -c -g -o $@ tmp.min.i
        rm -f tmp.i tmp.min.i
```

Here the preprocessor output is re-directed to a temporary file. Mincpp reads that temporary file and creates a second temporary file, which the compiler reads. Finally, both temporary files are deleted.

Most compiler drivers will recognize the file extension .i as already-preprocessed code and will skip the pass through cpp. When a fully implemented mincpp exists, it will be possible to use a "compiler driver-driver" that replaces cc. This obviates the need to modify makefiles. Another possibility is to use an environment variable that instructs cc as to which binary to use for the cpp pass. Many existing compiler drivers support this hook by using the variable cppC.

3.11 Implementation Summary

We have seen how `mincpp` processes the various syntactical structures of the C language. In order to keep the narrative rather straightforward, simple examples have been used. In the next chapter we will see that `mincpp` can process realistic examples based on complicated header files from the X window system and the various facilities provided by the UNIX system, as well as the facilities provided by a commercial database system vendor.

4. RESULTS

Compile times rise dramatically when using type-rich header files. Programs that use these include clients of Motif, Open Look, Microsoft Windows, application frameworks, and others. As applications and toolkits continue to grow in complexity, header files are likely to only grow further.

4.1 Benchmark Descriptions

`Mincpp` was evaluated by observing compile time and various other factors for a number of different benchmarks. These were chosen to represent code from different application areas, as well as having a variety of Nova factors. To review, Nova is the number of bytes after preprocessing divided by the number of bytes in the source module originally (`.i` / `.c`).

The benchmarks are:

1. `hello, world`. The classic first C program from Kernighan & Ritchie [Kernighan88].
2. `xmtravel`, from the X/Motif version 1.2.2 demo directory. This is a mockup of a sample "travel agency" application that tracks airline reservations.
3. `wc` (word count program) from the Berkeley BSD 4.3 sources.
4. `tims` (Technical Information Management System) a database library example program from Raima Data Manager 3.21a [Raima92].
5. `ol_fonts`, font-related applications from Sun Open Look version 3.

Note that none of the benchmarks contain global variables that cause unsafe `extern` minimizations as described earlier. So, the results presented here are the same with or without the option to disclose space-allocating declarations.

4.1.1 Small Benchmarks

The first three benchmarks, `hello`, `wc`, and `tims`, will be collectively referred to as the *small* benchmarks. These benchmarks have smaller file size increases after preprocessing that do the others. As we will see, some compile time improvements are obtainable even for the small cases, but the elapsed times are too small to really matter.

4.1.2 Large Benchmarks

`Xmtravel` and `ol_fonts` are referred to as the *large* benchmarks. This is actually a misnomer since both are demonstration programs that are quite modest in size compared to a typical application that one might develop. The common thread here is the use of the X Windows system. As we will see, the larger the include files, the more compile times can be reduced.

4.2 Directory Structure

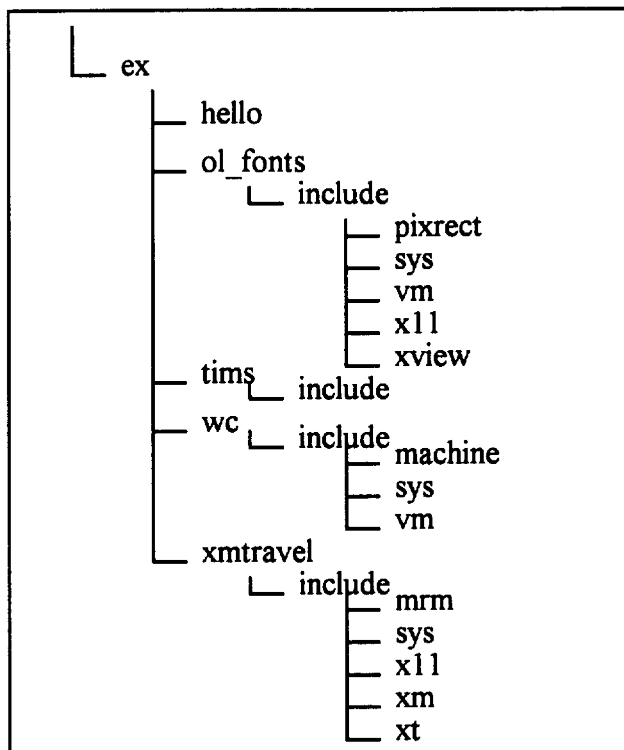

Figure 4.1 Examples Directory Structure

Figure 4.1 shows the layout of directories under the examples directory. The source code for the various benchmarks was assembled under one directory for easier experiment management. The standard preprocessor was run for each C module to scan for path names of include files. This list was used to create a local include directory under each benchmark's directory. Copies of all the header files needed in preprocessing were made locally. This simplifies the task of creating a common `makefile` (just use `-I.` in the rule) and also allows for easier transport of the directory across different machines.

4.3 Header File Structures

A tool to help understand header file include dependencies was constructed. This tool takes a `.i` file with embedded line directives (`#line`) and creates a hierarchically-organized list of the files. A C program then generates PostScript code that shows the

`#include` file structure graphically. Following are text-based hierarchical include lists for four of the benchmark modules, with graphical renderings of three:

```
// include hierarchy for tims delinfo.c benchmark
//
  1 delinfo.c
  2    include/stdio.h
  3    include/vista.h
  4       include/dproto.h
  5       include/lockcomm.h
  6    ./tims.h
```

```
// include hierarchy for wc.c benchmark
//
  1 wc.c
  2    include/sys/param.h
  3       include/machine/param.h
  4          include/machine/devaddr.h
  5       include/sys/signal.h
  6          include/vm/faultcode.h
  7          include/sys/stdtypes.h
  8       include/sys/types.h
  9          include/sys/stdtypes.h
 10          include/sys/sysmacros.h
 11    include/sys/stat.h
 12       include/sys/types.h
 13    include/sys/file.h
 14       include/sys/types.h
 15       include/sys/fcntlcom.h
 16          include/sys/stdtypes.h
 17          include/sys/stat.h
 18    include/stdio.h
```

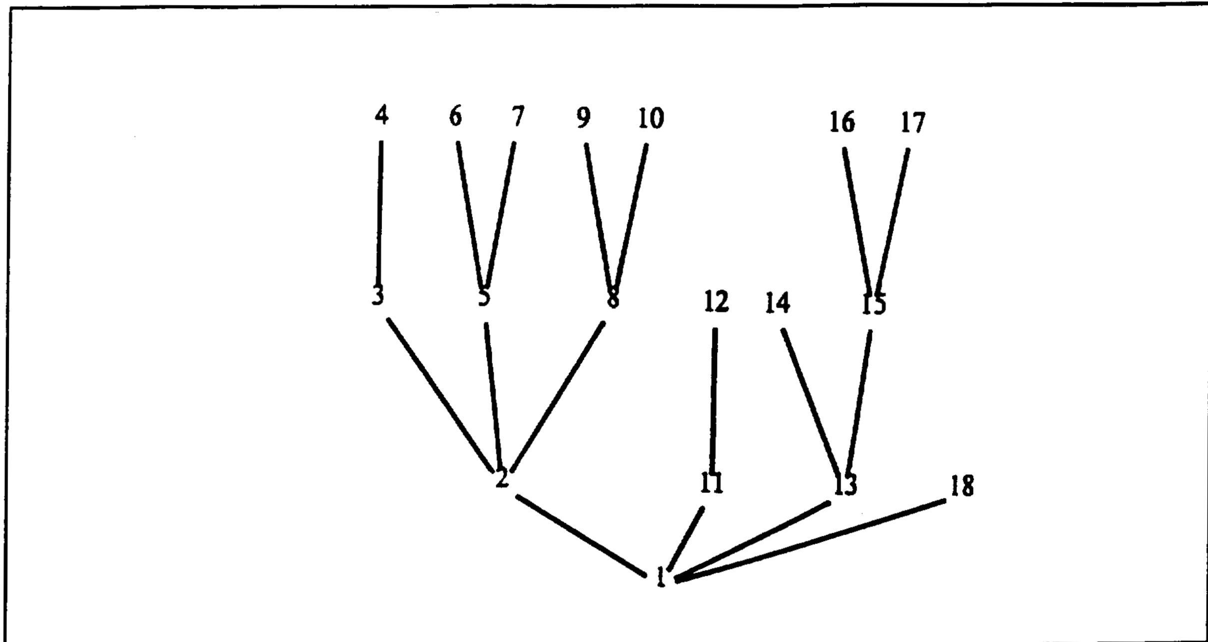

Figure 4.2 wc.c Include Tree

```
// include hierarchy for xmtravel trip_cb.c benchmark
//
  1 trip_cb.c
  2    include/Xm/Xm.h
  3       include/X11/Intrinsic.h
  4          include/X11/Xlib.h
  5             include/sys/types.h
  6                include/sys/stdtypes.h
  7                include/sys/sysmacros.h
  8             include/X11/X.h
  9             include/X11/Xfuncproto.h
 10             include/X11/Xosdefs.h
 11             include/stddef.h
 12                include/sys/stdtypes.h
 13          include/X11/Xutil.h
 14          include/X11/Xresource.h
 15          include/X11/Xfuncproto.h
 16          include/X11/Xosdefs.h
 17          include/string.h
 18             include/sys/stdtypes.h
 19          include/X11/Core.h
 20          include/X11/Composite.h
 21          include/X11/Constraint.h
 22          include/X11/Object.h
 23          include/X11/RectObj.h
 24       include/X11/Shell.h
 25       include/X11/Xatom.h
 26       include/Xm/XmStrDefs.h
 27          include/X11/StringDefs.h
 28       include/Xm/VirtKeys.h
 29          include/Xm/Xm.h
 30       include/Xm/VendorS.h
 31          include/Xm/Xm.h
```

```
32    include/Xm/DialogS.h
33       include/Xm/Xm.h
34    include/Xm/SelectioB.h
35       include/Xm/Xm.h
36    include/Xm/MessageB.h
37       include/Xm/Xm.h
38    include/Xm/Text.h
39       include/Xm/Xm.h
40       include/stdio.h
41    ./xmtravel.h
42       include/stdio.h
43       include/Xm/Xm.h
44       include/Mrm/MrmPublic.h
45          include/X11/Intrinsic.h
46          include/Mrm/MrmDecls.h
47       ./menu_cb.h
48       ./client_cb.h
49       ./trip_cb.h
50       ./dialog.h
```

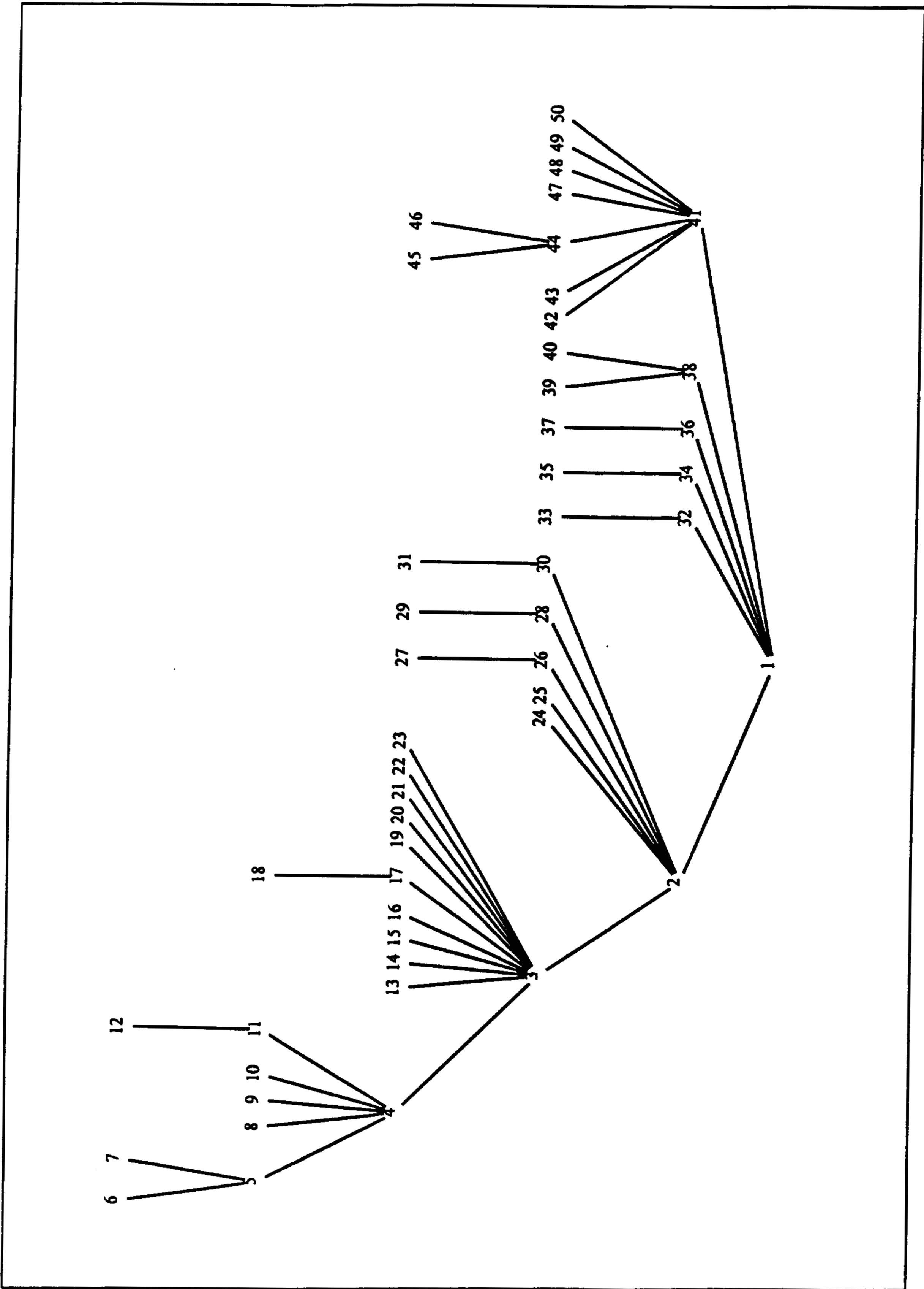

Figure 4.3 `trip_cb.c` Include Tree

```
// include hierarchy for ol_fonts fonts.c benchmark
  1 fonts.c        2    include/ctype.h
  3       include/c_varieties.h
  4    include/X11/X.h
  5    include/X11/Xlib.h
  6       include/sys/types.h
  7          include/sys/stdtypes.h
  8          include/sys/sysmacros.h
  9       include/X11/X.h
 10    include/xview/xview.h
 11       include/xview/xview_xvin.h
 12          include/signal.h
 13             include/c_varieties.h
 14             include/sys/signal.h
 15                include/vm/faultcode.h
 16                include/sys/stdtypes.h
 17          include/sys/types.h
 18          include/pixrect/pixrect.h
 19             include/sys/types.h
 20          include/pixrect/pr_planegroups.h
 21          include/pixrect/pr_util.h
 22          include/pixrect/memvar.h
 23          include/pixrect/pixfont.h
 24          include/pixrect/traprop.h
 25          include/pixrect/pr_line.h
 26          include/xview/xv_c_types.h
 27          include/xview/generic.h
 28             include/xview/pkg_public.h
 29                include/xview/pkg.h
 30                   include/xview/attr.h
 31                      include/sys/types.h
 32                      include/xview/base.h
 33                         include/string.h
 34                            include/c_varieties.h
 35                            include/sys/stdtypes.h
 36                            include/memory.h
 37                               include/c_varieties.h
 38                         include/malloc.h
 39                            include/c_varieties.h
 40                            include/sys/stdtypes.h
 41                            include/alloca.h
 42                               include/c_varieties.h
 43                         include/xview/xv_c_types.h
 44                   include/xview/notify.h
 45                      include/stdio.h
 46                         include/c_varieties.h
 47                      include/sys/types.h
 48                      include/sys/wait.h
 49                         include/sys/stdtypes.h
 50                      include/sys/time.h
 51                         include/time.h
 52                            include/c_varieties.h
 53                            include/sys/types.h
 54                      include/sys/resource.h
 55                      include/xview/base.h
 56                   include/varargs.h
 57                      include/c_varieties.h
 58                include/xview/xv_error.h
 59                   include/xview/attr.h
 60                   include/xview/sun.h
```

```
 61                  include/xview/base.h
 62                  include/sys/types.h
 63                  include/stdio.h
 64            include/xview/server.h
 65               include/xview/generic.h
 66            include/xview/screen.h
 67               include/xview/generic.h
 68            include/xview/notify.h
 69            include/xview/pixwin.h
 70               include/pixrect/pixrect.h
 71               include/xview/rect.h
 72                  include/xview/xv_c_types.h
 73               include/xview/rectlist.h
 74                  include/xview/xv_c_types.h
 75               include/xview/base.h
 76            include/xview/win_input.h
 77               include/sys/types.h
 78               include/sys/time.h
 79               include/X11/Xlib.h
 80               include/xview/base.h
 81               include/xview/win_event.h
 82         include/xview/xv_version.h
 83         include/xview/icon.h
 84            include/xview/window.h
 85               include/xview/generic.h
 86               include/xview/server.h
 87               include/xview/screen.h
 88               include/xview/drawable.h
 89                  include/xview/generic.h
 90               include/xview/win_input.h
 91               include/xview/rect.h
 92               include/X11/Xlib.h
 93            include/xview/attrol.h
 94               include/xview/base.h
 95               include/xview/attr.h
 96            include/xview/svrimage.h
 97               include/xview/generic.h
 98               include/xview/pkg.h
 99               include/xview/drawable.h
100               include/sys/types.h
101               include/pixrect/pixrect.h
102         include/xview/frame.h
103            include/xview/window.h
104            include/xview/attrol.h
105            include/X11/X.h
106         include/xview/openmenu.h
107            include/xview/generic.h
108            include/xview/pkg.h
109            include/xview/attrol.h
110            include/xview/window.h
111      include/xview/panel.h
112         include/xview/canvas.h
113            include/xview/openwin.h
114               include/xview/window.h
115               include/xview/attrol.h
116            include/xview/pixwin.h
117            include/xview/win_input.h
118         include/xview/frame.h
119      include/xview/font.h
120         include/xview/generic.h
```

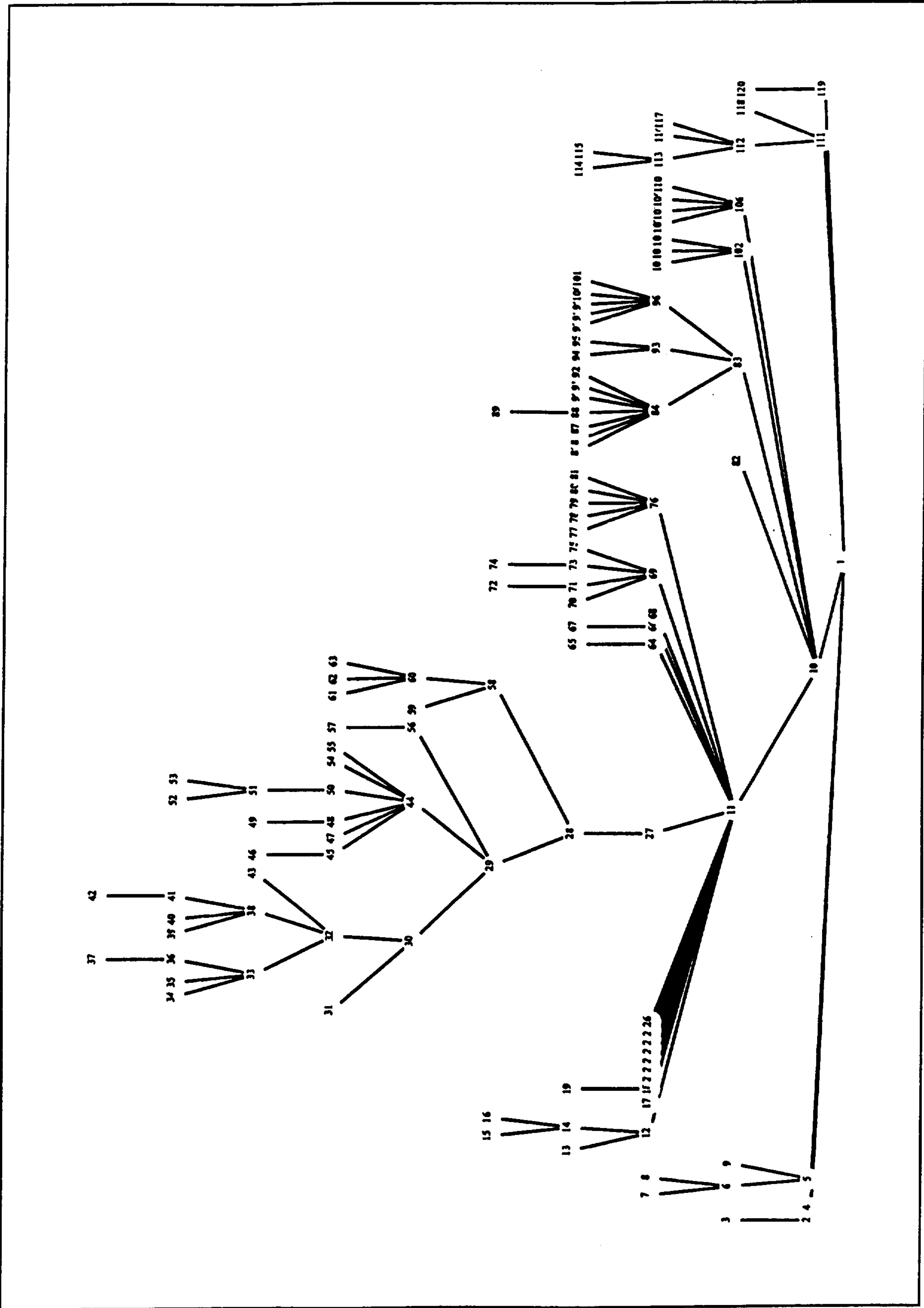

Figure 4.4 `fonts.c` Include Tree

4.4 Nocom

A tool called nocom was constructed. Nocom is simply a lex application with a small driver that tokenizes a file and strips white-space. The primary goal of constructing this tool was to put to rest any uncertainty as to whether or not all we were doing was simple file stripping.

In a practical application, blank lines are not removed so that subsequent compilation passes can report accurate line number information to the user. Here, nocom also strips blank lines so that we can very accurately see the decrease in lexically significant tokens that mincpp reduces. The makefile runs cpp to create the .i file. These are stripped by nocom to create a .nc.i file ("no comments"). Mincpp is run on the .nc.i to create a .min.i ("minimized").

4.5 Makefiles and Build Scripts

One of the advantages of collecting all the source in a common area is that a consistent build environment can be created. For our experiments, we use common formats and conventions, and then include a common makefile fragment that contains all the translation rules (also referred to as suffix rules or pattern rules). This ensures that build transcripts are in a uniform format, which in turn allows automated creation of tables and charts possible.

GNU make is used, primarily to access two features not supplied with standard make: pattern rules and include-able makefiles. The pattern rule syntax is considerably more flexible and powerful that the standard make suffix rule (.c.i:), while the included makefile facility allows common definitions to be stored in one place and used in multiple makefiles. For readers not familiar with gnu make, the suffix rule .c.o expressed as a pattern rule is %.o : %.c, and the syntax "include file" works just like a #include in C source.

Xmtravel's makefile follows:

```
OBJS = client_cb.o dialog.o menu_cb.o trip_cb.o xmtravel.o
IFILES = client_cb.i dialog.i menu_cb.i trip_cb.i xmtravel.i
```

```
DBGOBJS = client_cb.dbg.o dialog.dbg.o menu_cb.dbg.o trip_cb.dbg.o
xmtravel.dbg.o
OPTOBJS = client_cb.opt.o dialog.opt.o menu_cb.opt.o trip_cb.opt.o
xmtravel.opt.o
MINOBJS = client_cb.min.o dialog.min.o menu_cb.min.o trip_cb.min.o
xmtravel.min.o
DBGMINOBJS = client_cb.dbg.min.o dialog.dbg.min.o menu_cb.dbg.min.o
trip_cb.dbg.min.o xmtravel.dbg.min.o
OPTMINOBJS = client_cb.opt.min.o dialog.opt.min.o menu_cb.opt.min.o
trip_cb.opt.min.o xmtravel.opt.min.o

EXE = xmtravel xmtravel.min xmtravel.dbg xmtravel.dbg.min xmtravel.opt
xmtravel.opt.min

EXTRA_LIBS=-lMrm -lXm -lXt -lX11

include ../mkdef

xmtravel: $(OBJS)
xmtravel.min: $(MINOBJS)
client_cb.o : client_cb.c
dialog.o : dialog.c
menu_cb.o : menu_cb.c
trip_cb.o : trip_cb.c
xmtravel.o: xmtravel.c
client_cb.min.o : client_cb.min.i
dialog.min.o : dialog.min.i
menu_cb.min.o : menu_cb.min.i
trip_cb.min.o : trip_cb.min.i
xmtravel.min.o: xmtravel.min.i
xmtravel.dbg : $(DBGOBJS)
xmtravel.dbg.min : $(DBGMINOBJS)
xmtravel.opt : $(OPTOBJS)
xmtravel.opt.min : $(OPTMINOBJS)
client_cb.dbg.o : client_cb.c
dialog.dbg.o : dialog.c
menu_cb.dbg.o : menu_cb.c
trip_cb.dbg.o : trip_cb.c
xmtravel.dbg.o: xmtravel.c
client_cb.opt.o : client_cb.c
dialog.opt.o : dialog.c
menu_cb.opt.o : menu_cb.c
trip_cb.opt.o : trip_cb.c
xmtravel.opt.o: xmtravel.c
client_cb.opt.min.o : client_cb.min.i
dialog.opt.min.o : dialog.min.i
menu_cb.opt.min.o : menu_cb.min.i
trip_cb.opt.min.o : trip_cb.min.i
xmtravel.opt.min.o: xmtravel.min.i
client_cb.dbg.min.o : client_cb.min.i
dialog.dbg.min.o : dialog.min.i
menu_cb.dbg.min.o : menu_cb.min.i
trip_cb.dbg.min.o : trip_cb.min.i
xmtravel.dbg.min.o: xmtravel.min.i
client_cb.i : client_cb.c
dialog.i : dialog.c
menu_cb.i : menu_cb.c
trip_cb.i : trip_cb.c
xmtravel.i: xmtravel.c
client_cb.min.i : client_cb.nc.i
```

```
dialog.min.i : dialog.nc.i
menu_cb.min.i : menu_cb.nc.i
trip_cb.min.i : trip_cb.nc.i
xmtravel.min.i: xmtravel.nc.i
client_cb.nc.i : client_cb.i
dialog.nc.i : dialog.i
menu_cb.nc.i : menu_cb.i
trip_cb.nc.i : trip_cb.i
xmtravel.nc.i: xmtravel.i
```

In the `makefile` above, the first lines show groups of files that are the object files that comprise a single executable. Lines of the form "`<text> : <text>`" show build order dependencies: a translation including the file to the right of the colon results in the creation of the file on the left.

Every module follows a certain translation flow. The `.c` is read by `cc` to create three flavors of `.o` files, `.o`, `.dbg.o`, and `.opt.o`. These correspond to compiling with switches `-c`, `-g`, and `-O` respectively. The `.c` is also read by `cpp` which creates the `.i`. The `.i` is read by `nocom` to create the `.nc.i`. The `.nc.i` is read by `mincpp` to create the `.min.i`. The `.min.i` is read by `cc` three times to create the three flavors of minimized `.o`'s. There is a single executable (`xmtravel`) created from the five separate `.o` modules. There are six flavors of compile overall, three each for standard/minimized corresponding to the three flavors (`-c`, `-g`, and `-O`).

While we collect data for all three build flavors, we look in detail at the debuggable flavor (`.dbg`, corresponding to `-g`). This assumption is driven by the observation that `mincpp` is most applicable to the development phase where the edit/compile/link/debug loop is focused on the creation and early testing of source. Here, full debug is the prevalent mode of compilation. One might drop symbols later (the vanilla flavor, no optimization, no symbols) and finally use optimization for release builds (and for pre-release builds to test for compiler optimization bugs!). However, we will see that `mincpp` reduces compile times even on optimized code.

`Mkdef` is a file included in every `makefile` to give a uniform transcript format and ensure consistency in build procedures. The rules in `mkdef` drive the entire build cycle: preprocess, compile, link, run (through the debugger). Testing was done by running applications manually and performing whatever operations with the executable were

deemed complete to show that there were no runtime variances. This was done informally: no attempt at evaluating testing effectiveness using test coverage[19] or runtime error checking tools[20] was done.

The complete text of `mkdef` and detailed descriptions of every translation rule will follow:

```
#CC=cc
CC=acc -D__STDC__
#CC=CC -D__STDC__ -D__cplusplus

#CPP=../../app/cccp -U__STDC__
CPP=../../app/cccp -D__STDC__
#CPP=../../app/cccp -D__STDC__ -D__cplusplus

%.min.i : %.nc.i
	USE_TRACE_CACHE=n; PRINT_TO_MEMORY=n; YYDEBUG=; \
	export USE_TRACE_CACHE; export YYDEBUG; export
PRINT_TO_MEMORY; \
	cp $< /tmp/; \
	time ../../c_src/mincpp < /tmp/$< > $@; \
	ls -l Trace; \
	USE_TRACE_CACHE=y; PRINT_TO_MEMORY=n; \
	cp Trace /tmp/; \
	time ../../c_src/mincpp > /dev/null; \
	rm Trace; \
	ls -l $@

%.nc.i : %.i
	time ../../c_src/nocom/nocom $< > foo; mv foo $@
	ls -l $@

%.i : %.c
	-time $(CPP) -P -I. -Iinclude $< > $@
	ls -l $@ $<
	time wc -c $@

%.opt.min.o : %.min.i
	time $(CC) -c -O -o $@ -I. -Iinclude $<
	ls -l $@

%.dbg.min.o : %.min.i
	echo "# 1 \"foo.c\"" > foo; \
	cat foo $< > tmp.i
	time $(CC) -c -g -o $@ tmp.i
	ls -l $@
	rm -f foo

%.min.o : %.min.i
	time $(CC) -c -o $@ -I. -Iinclude $<
```

---

[19] `tcov` or TestCenter [CenterLine93a]

[20] Purify, Sentinel, or InSight

```
        ls -l $@


%.opt.o : %.c
        time $(CC) -c -O -o $@ -I. -Iinclude $<
        ls -l $@

%.dbg.o : %.c
        time $(CC) -c -g -o $@ -I. -Iinclude $<
        ls -l $@

%.o : %.c
        time $(CC) -c -o $@ -I. -Iinclude $<
        ls -l $@


%.dbg.min : %.dbg.min.o
%.dbg : %.dbg.o
%.opt.min : %.opt.min.o
%.opt : %.opt.o
%.min : %.min.o
% : %.o
        time cc -o $@ $^ -Bstatic $(EXTRA_LIBS)
        if [ ! -x $@ ]; then rm -f $@; fi
        ../dbxit $@
        ls -l $@
        rm -f $@
        rm -f $^

all : $(EXE)

clean:
        rm -f core *.o *.i $(EXE) #*# *~
```

Table 4.1 Translation Rules

Rule	Description
`%.i : %.c`	`-time $(CPP) -P -I. -Iinclude $< > $@` The GNU preprocessor, `cccp`, is run, using the local directories as search paths for `#include` files. The switch `-P` specifies that `#line` number directives NOT be present in the output. `Mincpp` is not equipped to deal with these. The rule uses the automatic `make` variables `$<` and `$@` to specify the matching `.c` and `.i` files respectively.

%.nc.i : %.i	time ../../c_src/nocom/nocom $< > foo; mv foo $@ Nocom from the mincpp source directory (c_src) is run on the .i file and moved (through a temporary file) to the matching .nc.i file.
%.min.i : %.nc.i	USE_TRACE_CACHE=n; PRINT_TO_MEMORY=n; YYDEBUG=; \\ export USE_TRACE_CACHE; export YYDEBUG; export PRINT_TO_MEMORY; \\ cp $< /home/tmp/; time wc -c /home/tmp/$<; time wc - c /home/tmp/$<; \\ time ../../c_src/mincpp < /home/tmp/$< > $@; \\ USE_TRACE_CACHE=y; PRINT_TO_MEMORY=y; \\ cp Trace /home/tmp/; time wc -c /home/tmp/Trace; time wc -c /home/tmp/Trace; \\ time ../../c_src/mincpp; \\ rm Trace; \\ Several pieces of data are collected here. First, some environment variables are explicitly set and exported. **USE_TRACE_CACHE** specifies mincpp's mode: parse or replay. **PRINT_TO_MEMORY** is an un-implemented feature that was intended to allow mincpp to output to an in-memory data structure to measure file I/O effects. **YYDEBUG** is a yacc variable that allows rule reductions to be transcripted to stderr: this is a parser debugging tool. Wc is then run on a copy of the .i file. Mincpp is run in parse mode, then the resulting Trace file is examined for size, then mincpp is run in replay mode, then some clean-up occurs. Wc is run to allow measurement of file I/O overhead.

%.o : %.c
%.dbg.o : %.c
%.opt.o : %.c
 	
%.min.o :
%.min.i
%.dbg.min.o :
	%.opt.min.o : %.min.i
	        time $(CC) -c -O -o $@ -I. -Iinclude $<
	%.dbg.min.o : %.min.i
	        echo "# 1 \\"foo.c\\"" > foo; \\
	        cat foo $< > tmp.i
	        time $(CC) -c -g -o $@ tmp.i
	        rm -f tmp.i foo
	%.min.o : %.min.i
	        time $(CC) -c -o $@ -I. -Iinclude $<
	
	%.opt.o : %.c
	        time $(CC) -c -O -o $@ -I. -Iinclude $<
	%.dbg.o : %.c

<table>
<tr><td>%.min.i

%.opt.min.o :

%.min.i</td><td>time $(CC) -c -g -o $@ -I. -Iinclude $<
%.o : %.c
time $(CC) -c -o $@ -I. -Iinclude $<

These are pretty straightforward: there are separate rules to make sure the .o's have different suffixes to match their flavor. The rules match suffix and use -c, -g, and -O appropriately. There is a bug in Sun acc so that if there are no line numbers it causes a compiler crash. That is why the operations involving #line 1 foo.c are there.</td></tr>
<tr><td>% : %.o

%.dbg : %.o

%.opt : %.o

%.min : %.min.o

%.dbg.min :

%.min.o

%.opt.min :

%.min.o</td><td>time $(CC) -o $@ $^ -lxview -lolgx -lX11
../libvistamu.a -lXt -lXm -lMrm
../dbxit $@

The link. All the libraries that are needed in any of the binaries are specified so one rule works for all links. Dbxit is an external script that runs the program under the dbx debugger. This loads the program.</td></tr>
<tr><td>clean</td><td>rm -f core *.o *.i

This cleans things up.</td></tr>
</table>

Mkdef uses an external script to invoke the debugger, dbx and collect data on debugger load times. It just loads the program and quits so the time has a relationship to the load time of the binary.

```
#!/bin/sh

time dbx <<!
debug $1
quit
!
```

4.6 Collection Methodology and Tools

There is a top level `makefile` in the examples directory that changes directory into each of the 6 example directories and does a `make all`. This causes all builds and runs to occur and transcript to `stdout`.

```
all :
        cd hello; make clean; rm -f build.*
        cd wc; make clean; rm -f build.*
        cd tims; make clean; rm -f build.*
        cd xmtravel; make clean; rm -f build.*
        cd ol_fonts; make clean; rm -f build.*
        cd hello; make > ../build.log 2>&1
        cd wc; make >> ../build.log 2>&1
        cd tims; make >> ../build.log 2>&1
        cd xmtravel; make >> ../build.log 2>&1
        cd ol_fonts; make >> ../build.log 2>&1
        #
        # ... repeat
        #
        cd hello; make clean; rm -f build.*
        cd wc; make clean; rm -f build.*
        cd tims; make clean; rm -f build.*
        cd xmtravel; make clean; rm -f build.*
        cd ol_fonts; make clean; rm -f build.*
        cd hello; make >> ../build.log 2>&1
        cd wc; make >> ../build.log 2>&1
        cd tims; make >> ../build.log 2>&1
        cd xmtravel; make >> ../build.log 2>&1
        cd ol_fonts; make >> ../build.log 2>&1
        ./mktab.sh
```

This was run three times with a `make clean` in-between runs to force a full re-build for each iteration. All of the output was sent to a log file. This raw compile data was processed into table format by an `awk` program. The `awk` program keeps arrays of data for each event of interest. Since three runs of data are in the same file, it updates each array entry three times. At the end, each data bucket is averaged, and printed in table form suitable for processing by a spreadsheet program. Excel spreadsheets were created by loading `build.txt` and creating derivative spreadsheets.

All data was collected on a Sun SparcServer 670MP with 5 gigabytes SCSI-2 disk, 192 megabytes of memory, running SunOS 4.1.2. The data was collected while the machine was in single-user mode to insure that there would be no interference from the network or other users.

4.7 Result Data

The data collected by the procedures outlined above will be presented. A brief description of the areas of interest will be followed by a table and one or two graphs. Table column header descriptions will follow the matching table. Discussion of the results will conclude each section.

4.7.1 File Sizes

The increase in file size after preprocessing, and the decrease in size possible through minimization is directly correlated to the compile time reductions. Here we will show the sizes of the various intermediate files that are introduced by the preprocessing steps.

Table 4.2 Benchmark File Sizes

module	.c	.i	.nc.i	.min.i	Increase	Decrease
hello.c	21	1026	859	27	1005	999
wc.c	2026	2368	1766	1025	342	1343
listkeys.c	179	6934	1416	150	6755	6784
listauth.c	253	7018	1455	182	6765	6836
tims.c	644	7311	1608	432	6667	6879
delinfo.c	769	7472	1707	521	6703	6951
borrow.c	803	7589	1820	613	6786	6976
entinfo.c	1146	7794	1923	720	6648	7074
findpubs.c	1557	8011	2020	835	6454	7176
xmtravel.c	2131	46749	34074	6171	44618	40578
menu_cb.c	2381	48992	35220	2210	46611	46782
trip_cb.c	3163	50873	36452	6105	47710	44768
client_cb.c	4731	52112	37791	8148	47381	43964
dialog.c	5791	52563	38388	9329	46772	43234
simple_font.c	917	58968	49574	16584	58051	42384
disp_fonts.c	1241	59333	49889	19666	58092	39667
disp_fonts2.c	1264	59356	49901	19669	58092	39687
type_font.c	1261	71359	60582	28149	70098	43210
fonts.c	3062	73036	61897	32121	69974	40915

- `.c`: Size in bytes of original `.c` module.
- `.i`: Size in bytes of unmodified preprocessor output.

- `.nc.i`: Size in bytes of white space and comment stripped preprocessor output.
- `.min.i`: Size in bytes of minimized, white space, and comment stripped preprocessor output.
- Increase: Byte increase in size from `.c` file to `.i` file (`.i - .c`).
- Decrease: Byte decrease from `.i` file to `.min.i` file (`.i - .min.i`).

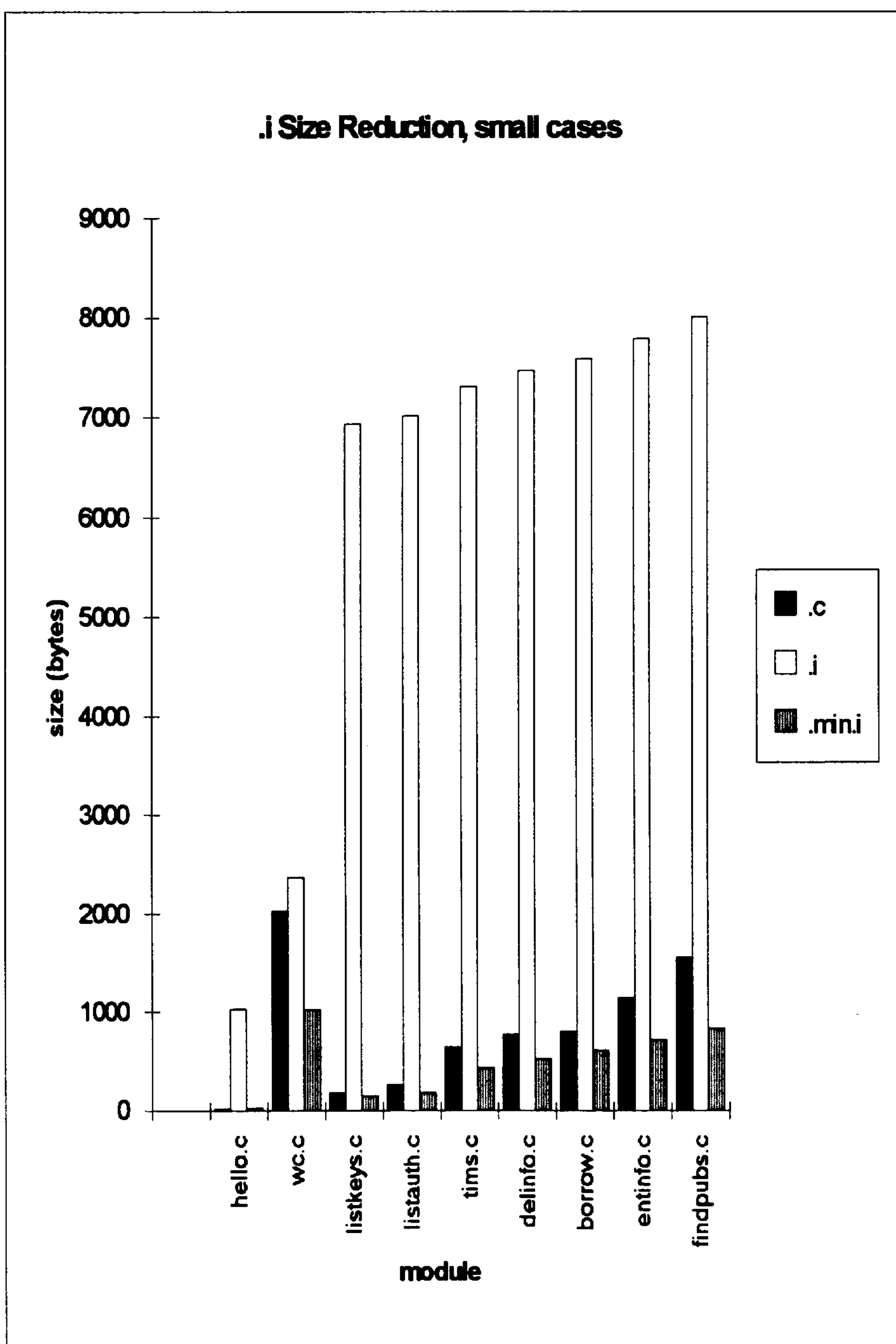

Figure 4.5 File Sizes: Small Cases

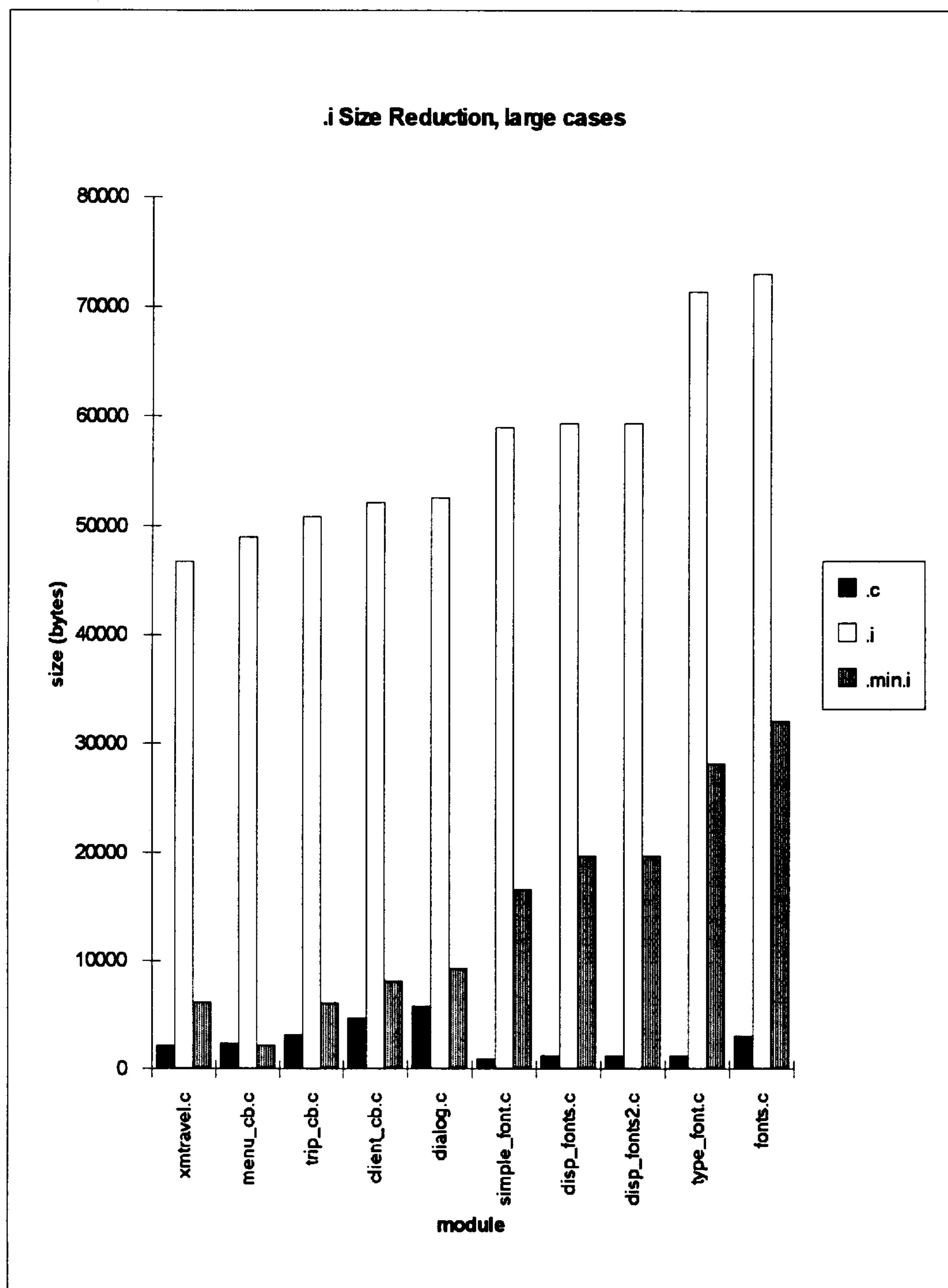

Figure 4.6 File Sizes: Large Cases

Here we show the sizes of files at the end of various preprocessing steps. There is a general trend for the `.min.i` to be around the same size as the original `.c` for the small

cases, around twice as large for the xmtravel modules, and around 10 times as large for the `ol_fonts` modules.

Following the discussion of the last chapter, it can be concluded that the font applications make more use of their header files than the other applications. Since more declarations are referenced, less text is spurious, and the minimization is comparatively less than for the other applications considered as a whole.

4.7.2 Cpp and Mincpp Times

The time to execute the preprocessing tools is critical in reducing compile times. Adding minimization to preprocessing must save more time than it adds to the preprocessing phase.

Table 4.3 Cpp/Mincpp Timings

module	cccp	wc	nocom	mincpp/s	mincpp/r	# inc	Trace	Worst Case
hello.c	0.10	0.10	0.10	0.10	0.00	1	5054	0.10
wc.c	0.20	0.10	0.10	0.20	0.00	17	10972	0.20
listkeys.c	0.20	0.10	0.10	0.13	0.00	5	11632	0.13
listauth.c	0.20	0.10	0.10	0.20	0.00	6	11883	0.20
tims.c	0.20	0.10	0.10	0.20	0.10	4	12646	0.20
delinfo.c	0.20	0.10	0.10	0.20	0.10	5	13091	0.20
borrow.c	0.20	0.10	0.10	0.20	0.10	5	13897	0.20
entinfo.c	0.20	0.10	0.10	0.20	0.10	5	14637	0.20
findpubs.c	0.20	0.10	0.20	0.20	0.10	5	15079	0.20
xmtravel.c	1.53	0.20	1.10	1.80	0.50	39	195117	2.13
menu_cb.c	1.40	0.20	1.20	1.90	0.50	42	200702	2.10
trip_cb.c	1.60	0.20	1.20	2.00	0.50	49	218289	2.40
client_cb.c	1.63	0.20	1.23	2.13	0.60	49	230755	2.56
dialog.c	1.80	0.20	1.30	2.20	0.60	56	231712	2.80
simple_font.c	2.10	0.30	1.50	3.57	0.60	118	249498	3.87
disp_fonts.c	2.10	0.30	1.50	3.57	0.60	117	250745	3.87
disp_fonts2.c	2.10	0.30	1.50	3.57	0.60	117	250795	3.87
type_font.c	2.30	0.30	1.80	4.50	0.70	119	310743	5.00
fonts.c	2.30	0.30	1.80	4.60	0.80	119	324382	5.10

- `cccp:` Time in seconds for cpp to run.

- `wc:` Time in seconds for `wc` to run.
- `nocom:` Time in seconds for `nocom` to run.
- `mincpp/s:` Time in seconds for `mincpp` to run in save mode. `Mincpp` is creating a `Trace` file and parsing 100% of its input.
- `mincpp/r:` Time in seconds for `mincpp` to run in replay mode. `Mincpp` is reading the `Trace` file and is parsing none of its input.
- # inc: Number of header files read in by the preprocessor.
- `Trace:` Size in bytes of `mincpp`'s trace file, created during save mode processing.
- Worst Case: Time in seconds for total preprocessing assuming no optimizations. This is `cpp` time plus `mincpp` time.

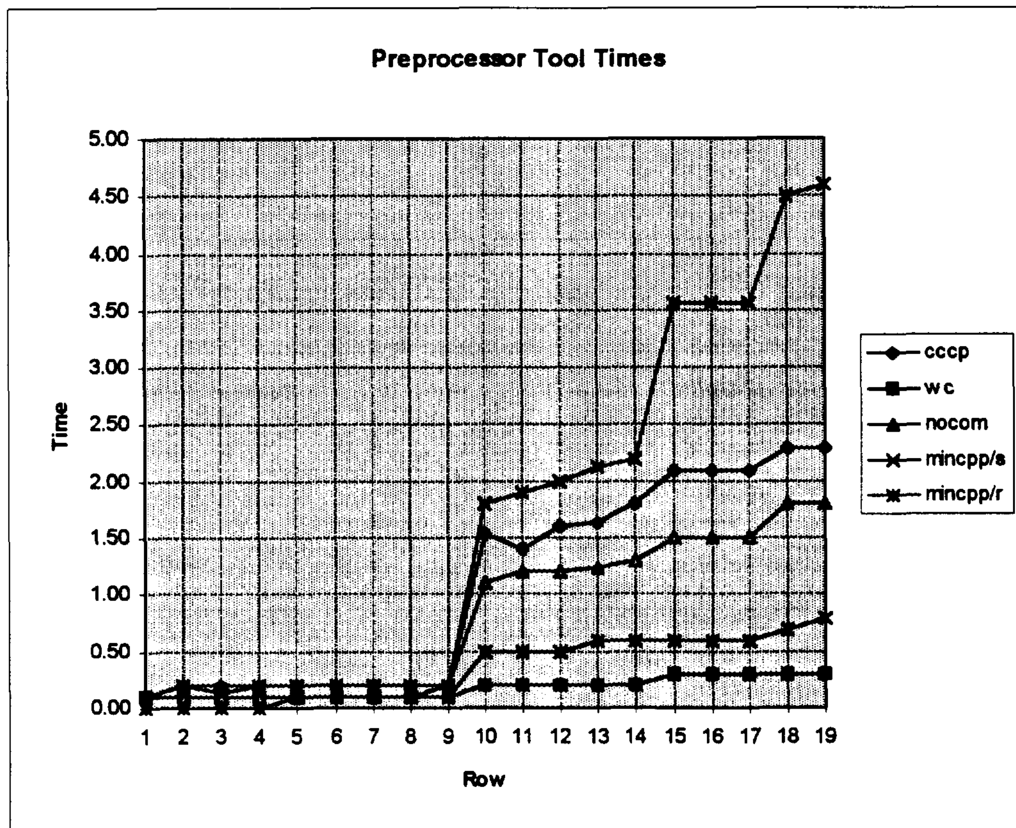

Figure 4.7 Cpp/Mincpp Timings Graph

We observe the `cpp` and `nocom` take roughly the same time to run. This is intuitively appealing because both tools are scanning the input and doing some processing based on token content. They are not parsing the source language but are instead

concerned with token manipulation: cpp with # directive interpretation and macro identifier identification, nocom with identifying white space tokens and matching the end of the white space.

Wc is given to show the raw time it takes to read the file off disk. This is used later to factor out some file I/O in calculating worst case timings for a cpp/mincpp joined binary.

Mincpp takes about the same time as cpp for the small cases, about 50% more for xmtravel, and about 2 times more for ol_fonts. Since mincpp includes a complete C parser, we do expect it to take more time that cpp. Some of this difference lies in relationship to how much code is stripped. The font code is stripped relatively less than the xmtravel code, so the output is bigger.

4.7.3 Compile Times

The most important data collected is on compile times. This table shows raw compile times for standard and minimized text. These numbers are too optimistic for the minimized case since no preprocessing is included. We need to add in preprocessing time. This is done in a later section considering three different preprocessing scenarios.

Table 4.4 Compile Times

module	cc(-g)	cc(-g/m)	cc	cc(m)	cc(-O)	cc(-O/m)
hello.c	0.60	0.60	0.60	0.50	0.90	0.80
wc.c	1.50	1.20	1.30	1.00	2.40	2.10
listkeys.c	0.90	0.60	0.80	0.60	1.20	1.00
listauth.c	0.90	0.63	0.80	0.60	1.30	1.00
tims.c	1.00	0.80	0.90	0.70	1.50	1.30
delinfo.c	1.10	0.80	0.90	0.73	1.60	1.30
borrow.c	1.10	0.90	1.00	0.73	1.67	1.40
entinfo.c	1.20	0.90	1.00	0.80	1.93	1.70
findpubs.c	1.37	1.00	1.10	0.80	2.00	1.80
xmtravel.c	5.03	1.60	4.10	1.20	4.60	1.73
menu_cb.c	5.23	1.20	4.30	1.00	5.53	2.10
trip_cb.c	5.73	2.00	4.90	1.50	6.10	2.60
client_cb.c	6.23	2.57	5.27	1.90	7.20	3.60
dialog.c	7.10	3.10	6.00	2.43	8.40	4.70
simple_font.c	7.93	2.03	6.77	1.80	7.33	2.40
disp_fonts.c	7.93	2.40	7.00	2.10	7.67	2.80
disp_fonts2.c	7.90	2.43	6.93	2.10	7.67	2.83
type_font.c	9.07	3.23	7.77	2.73	8.73	3.70
fonts.c	9.63	4.00	8.27	3.40	10.57	5.70

- cc(-g): Time in seconds for debug compile of unmodified module.
- cc(-g/m): Time in seconds for debug compile of minimized module.
- cc: Time in seconds for non-debug, non-optimized (-c) compile of unmodified module.
- cc(min): Time in seconds for non-debug, non-optimized (-c) compile of minimized module.
- cc(-O): Time in seconds for optimized compile of unmodified module.
- cc(-O/m): Time in seconds for optimized compile of minimized module.

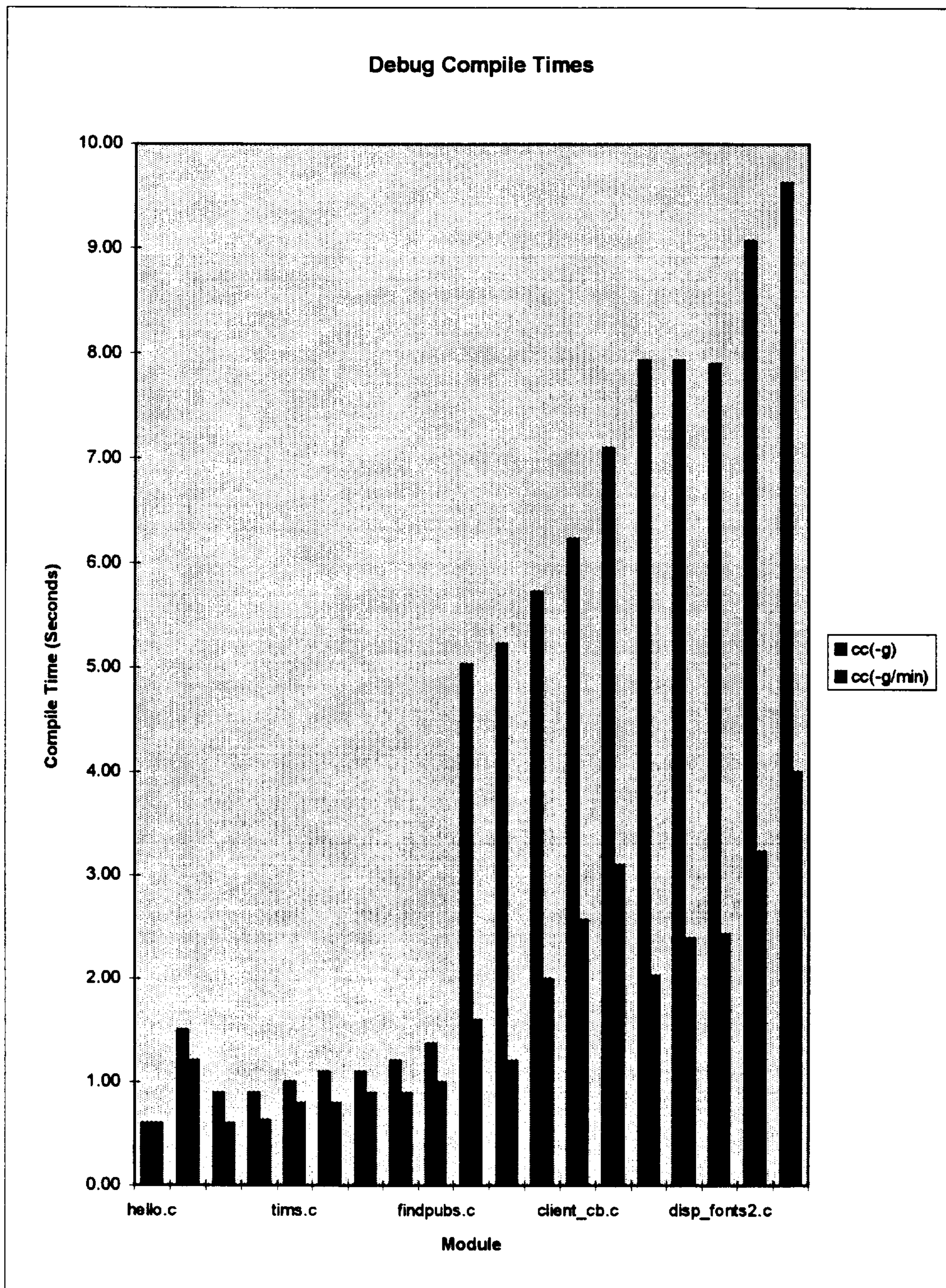

Figure 4.8 Compile Times Graph

The non-minimized are for straight `cc`, reading from the `.c` file. The minimized are for `cc` reading the `.min.i`, so these times are too optimistic since they factor out

preprocessing. A later table will show times for factoring this in several different forms based on the attributes of the build.

4.7.4 Object File Sizes

An interesting side-effect of minimization is that object file sizes for debug compiles are reduced. This is intuitively appealing, since we reason that the declarations that are missing are also not taking space in the object file's symbol table.

Table 4.5 Object File Sizes

module	.o(-g)	.o(-g/m)	.o	.o(m)	.o(-O)	.o(-O/m)
hello.c	1368	1068	168	168	152	152
wc.c	12452	9520	5088	4916	3804	3632
listkeys.c	3608	1940	764	764	712	712
listauth.c	3888	2220	916	916	876	876
tims.c	5148	4124	2580	2580	2492	2492
delinfo.c	5604	4164	2124	2124	1904	1904
borrow.c	6500	4964	2556	2556	2240	2240
entinfo.c	7532	6164	3436	3436	2996	2996
findpubs.c	8060	6620	3640	3640	3404	3404
xmtravel.c	55200	24828	6552	6152	6264	5864
menu_cb.c	55756	10324	4536	4128	3840	3432
trip_cb.c	60180	28580	7188	6772	5460	5044
client_cb.c	66588	36156	10392	10392	7724	7724
dialog.c	69536	37892	12296	11884	8176	7764
simple_font.c	62388	19080	1528	1528	1376	1376
disp_fonts.c	64432	23472	3216	3216	2884	2884
disp_fonts2.c	64528	23568	3180	3180	2816	2816
type_font.c	75064	37248	3284	3284	2732	2732
fonts.c	81940	45836	8480	8480	7100	7100

- .o(-g): Size in bytes of object file compiled debug using unmodified module.
- .o(-g/m): Size in bytes of object file compiled debug using minimized module.
- .o: Size in bytes of object file compiled non-debug, non-optimized (-c) using unmodified module.
- .o(min): Size in bytes of object file compiled non-debug, non-optimized (-c) using minimized module.

- .o(-O): Size in bytes of object file compiled optimized using unmodified module.
- .o(-O/m): Size in bytes of object file compiled optimized using minimized module.

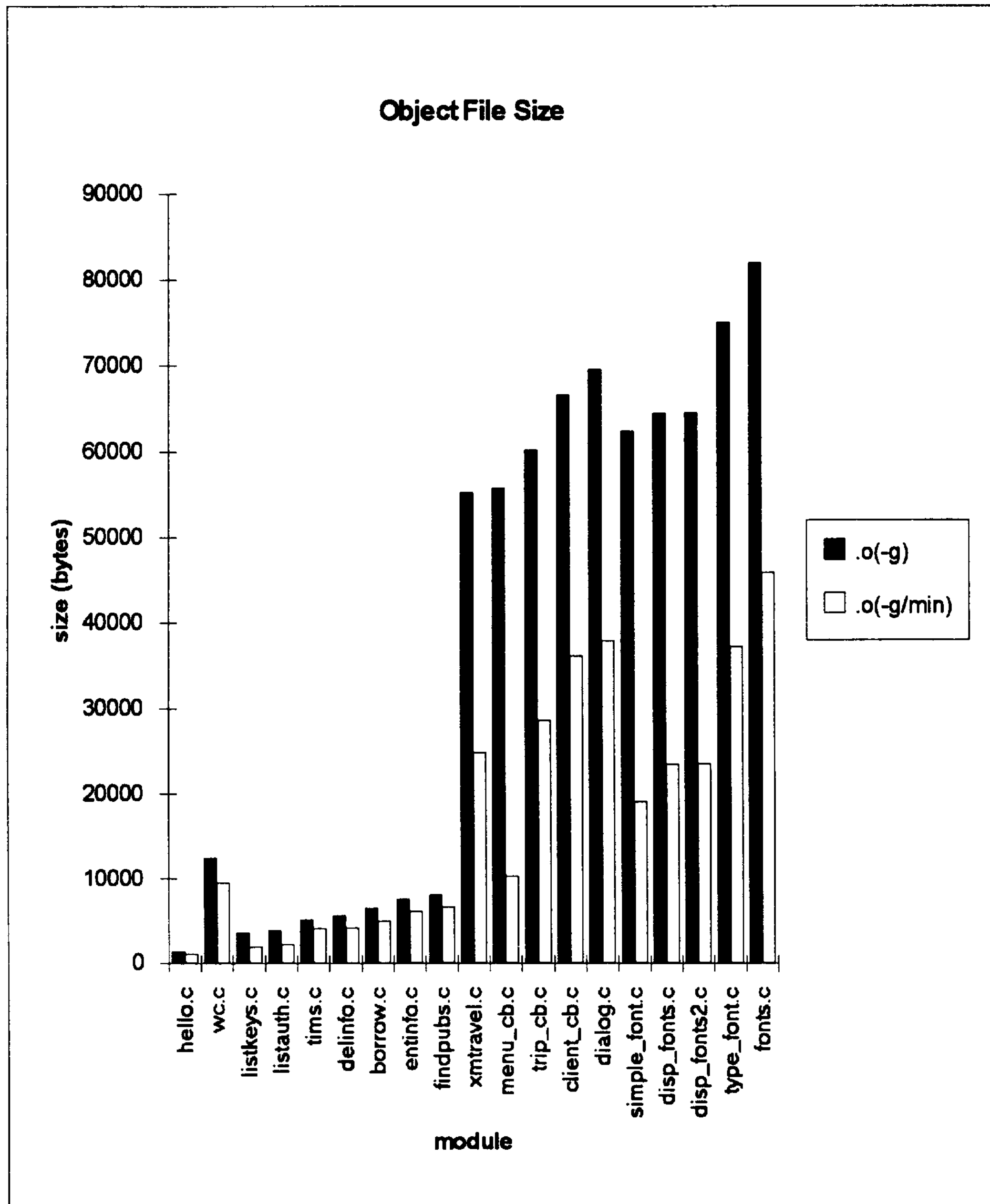

Figure 4.9 Object File Sizes

The interesting result here is that `mincpp` is able to reduce object file sizes considerably for debug compiles. This results in disk space savings as well as some incremental benefits in link and debugger load time. These are incremental benefits and

will not be discussed further to keep the discussion focused on compile time issues. The reduction in `.o` size is explained by the fact that the object file's symbol table has information for all the types mentioned in all header files, whether they are needed or not. Since `mincpp` is deleting unneeded declarations, the space that would have been needed for these is reclaimed. Again, the large examples show a greater difference than do the small examples.

4.7.5 Linker and Binary Results

The reductions in object file size for debug compiles also result in some decrease in linker and debugger load times. These are presented for the debug case only.

Table 4.6 Linker/Binary Statistics

binary	ld	ld min	syms	syms min	dbx	dbx min	a.out	a.out min
hello/-g	0.90	0.90	306	299	0.50	0.50	106496	106496
wc/-g	1.00	1.00	608	536	0.50	0.50	114688	114688
tims/-g	1.60	1.60	1427	1371	0.77	0.77	270336	270336
xmtravel/-g	9.60	9.37	9774	9284	4.60	4.57	2195456	2179072
simple_font/-g	8.50	8.30	5645	5014	3.13	3.03	1523712	1482752
disp_fonts/-g	8.57	8.37	5669	5055	3.17	3.03	1531904	1490944
disp_fonts2/-g	8.60	8.40	5680	5066	3.20	3.00	1531904	1490944
type_font/-g	8.50	8.37	5788	5185	3.17	3.10	1531904	1499136
fonts/-g	8.50	8.30	5893	5300	3.20	3.03	1540096	1507328
hello	0.90	0.90	276	276	0.50	0.47	106496	106496
wc	1.00	1.00	302	300	0.50	0.50	106496	106496
tims	1.60	1.57	708	708	0.70	0.70	262144	262144
xmtravel	9.27	9.20	6960	6956	4.07	4.03	2080768	2080768
simple_font	8.33	8.37	4824	4824	2.83	2.83	1458176	1458176
disp_fonts	8.40	8.40	4832	4832	2.80	2.83	1466368	1466368
disp_fonts2	8.33	8.37	4832	4832	2.83	2.90	1466368	1466368
type_font	8.33	8.37	4827	4827	2.83	2.87	1458176	1458176
fonts	8.33	8.33	4835	4835	2.80	2.80	1466368	1466368
hello/-O	0.90	0.90	276	276	0.47	0.43	106496	106496
wc/-O	1.00	1.00	302	300	0.50	0.50	106496	106496
tims/-O	1.57	1.60	708	708	0.70	0.70	262144	262144
xmtravel/-O	9.23	9.20	6960	6956	4.03	4.13	2072576	2072576
simple_font/-O	8.30	8.30	4824	4824	2.87	2.80	1458176	1458176
disp_fonts/-O	8.30	8.33	4832	4832	2.83	2.83	1466368	1466368
disp_fonts2/-O	8.30	8.30	4832	4832	2.83	2.83	1466368	1466368
type_font/-O	8.37	8.30	4827	4827	2.83	2.80	1458176	1458176
fonts/-O	8.30	8.30	4835	4835	2.83	2.87	1466368	1466368

- ld: Time in seconds to link the unmodified executable.
- ld min: Time in seconds to link the minimized executable.
- syms: Number of debug symbols in the executable as reported by the debugger for the unmodified executable.

- syms min: Number of debug symbols in the executable as reported by the debugger for the minimized executable.
- dbx: Time in seconds to load the unmodified executable into the debugger.
- dbx min: Time in seconds to load the minimized executable into the debugger.
- a.out: Size in bytes of the unmodified executable.
- a.out min: Size in bytes of the minimized executable.

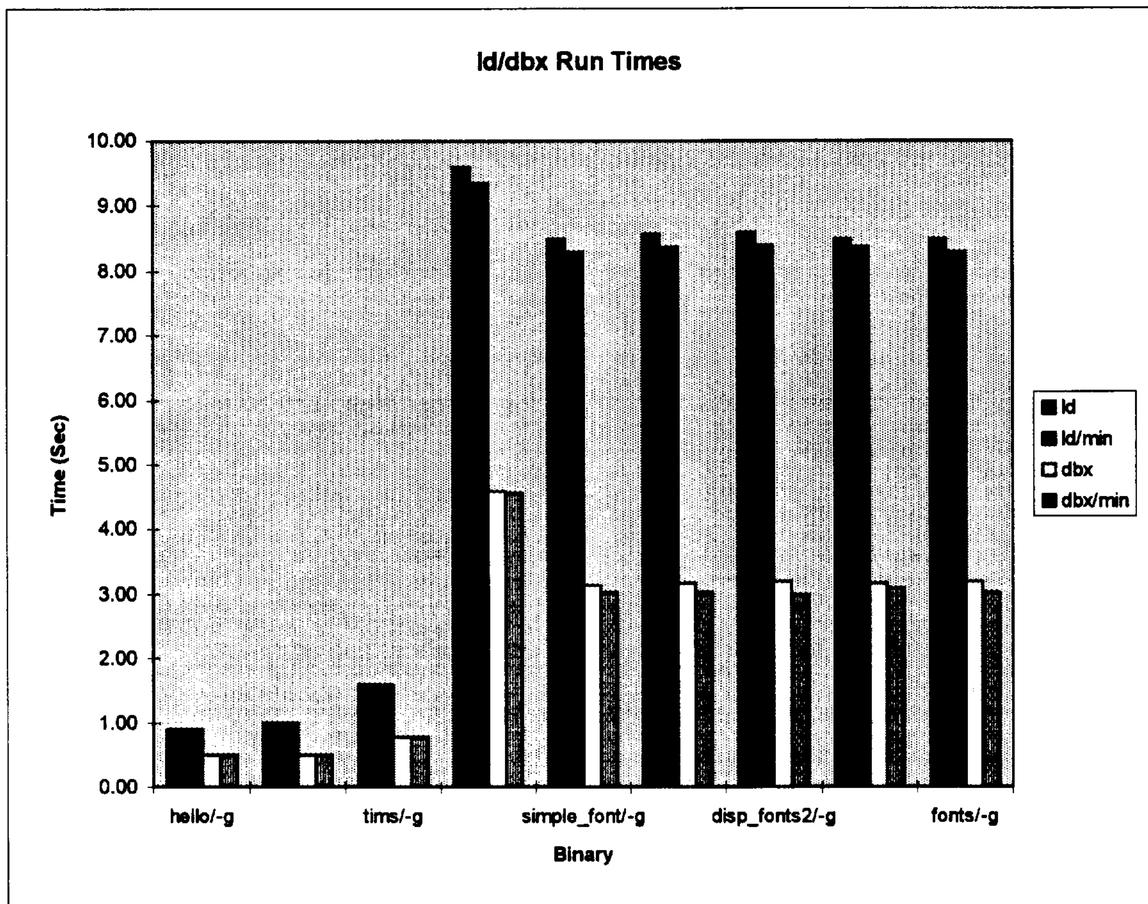

Figure 4.10 Linker/Debugger Graph

We observe some positive effects from minimization for debug built binaries, but not any real effect for the other two flavors. An interesting experiment to conduct would be to look at very large binaries with tens to hundreds of thousands of symbols and see if the effect increases.

4.7.6 Overall Compile Times

Here, we consider more realistic compile times. We use the standard compile time charted earlier, but now we add preprocessing time back into the minimized compile time. This is done three ways: full preprocessing plus minimization (minimization does full parsing), minimization with full replay, and minimization (full replay) where the build can be avoided. Because of the length of the column descriptions, they are given after the chart.

Table 4.7 Overall Compile Times

module	cc	worst case	100% cache build needed	build avoided	cccp	sav	cc min	rst	wc
hello.c	0.60	0.20	0.60	0.00	0.1	0.1	0.6	0	0.1
wc.c	1.50	1.00	1.20	0.00	0.2	0.2	1.2	0	0.1
listkeys.c	0.90	0.33	0.60	0.00	0.2	0.13	0.6	0	0.1
listauth.c	0.90	0.43	0.63	0.00	0.2	0.2	0.63	0	0.1
tims.c	1.00	0.60	0.90	0.10	0.2	0.2	0.8	0.1	0.1
delinfo.c	1.10	0.60	0.90	0.10	0.2	0.2	0.8	0.1	0.1
borrow.c	1.10	0.70	1.00	0.10	0.2	0.2	0.9	0.1	0.1
entinfo.c	1.20	0.70	1.00	0.10	0.2	0.2	0.9	0.1	0.1
findpubs.c	1.37	0.80	1.10	0.10	0.2	0.2	1	0.1	0.1
xmtravel.c	5.03	3.73	2.10	0.50	1.53	1.8	1.6	0.5	0.2
menu_cb.c	5.23	3.30	1.70	0.50	1.4	1.9	1.2	0.5	0.2
trip_cb.c	5.73	4.40	2.50	0.50	1.6	2	2	0.5	0.2
client_cb.c	6.23	5.13	3.17	0.60	1.63	2.13	2.57	0.6	0.2
dialog.c	7.10	5.90	3.70	0.60	1.8	2.2	3.1	0.6	0.2
simple_font.c	7.93	5.90	2.63	0.60	2.1	3.57	2.03	0.6	0.3
disp_fonts.c	7.93	6.27	3.00	0.60	2.1	3.57	2.4	0.6	0.3
disp_fonts2.c	7.90	6.30	3.03	0.60	2.1	3.57	2.43	0.6	0.3
type_font.c	9.07	8.23	3.93	0.70	2.3	4.5	3.23	0.7	0.3
fonts.c	9.63	9.10	4.80	0.80	2.3	4.6	4	0.8	0.3

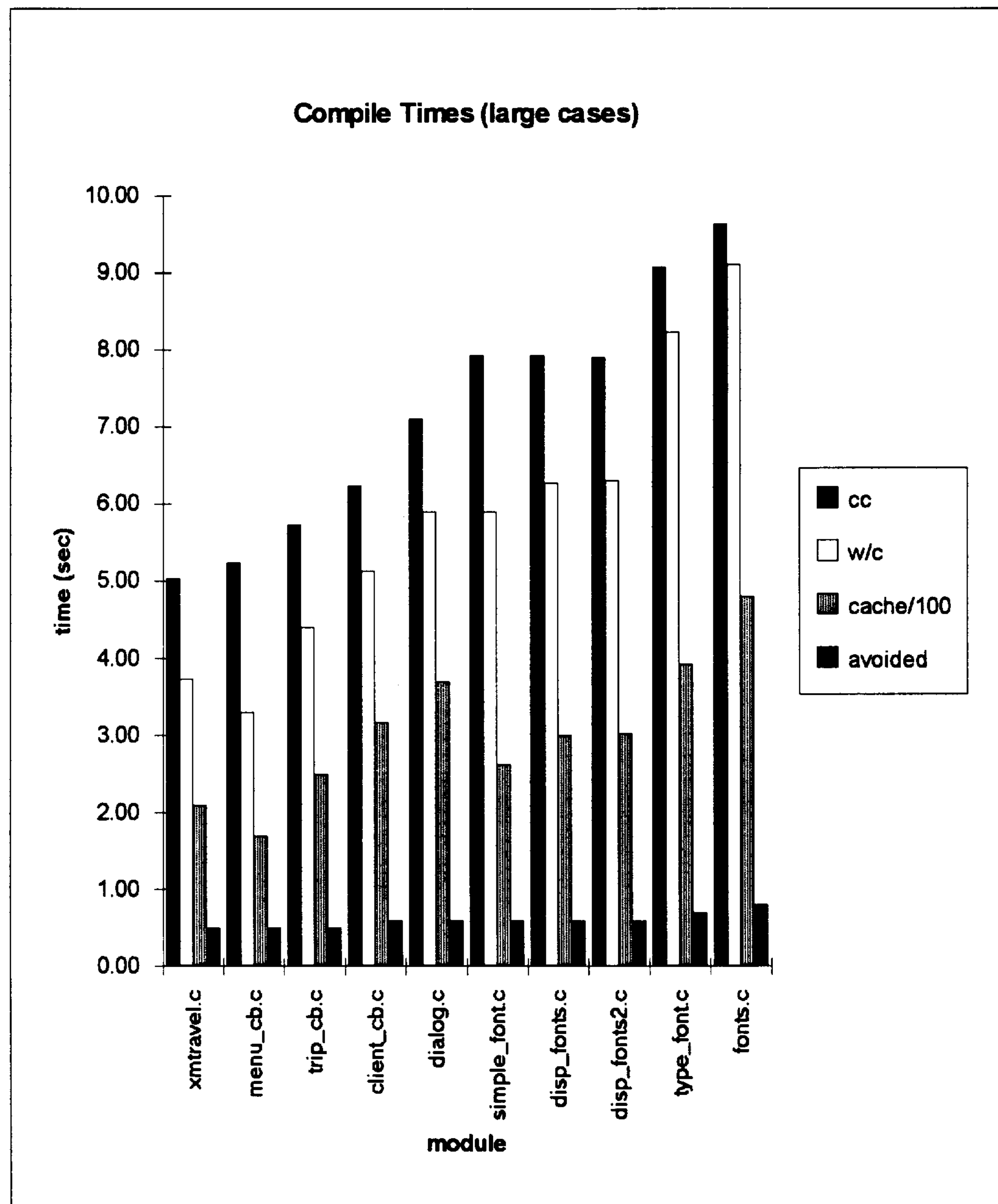

Figure 4.11 Overall Compile Times: Large Cases

We now proceed to describe the table column descriptions and discuss the methodology and results.

Table 4.8 Overall Compile Times Column Descriptions

Column	Description
cc	Standard compile using cc as compiler driver on the .c file.
worst case mincpp	Worst case performance with mincpp. Formula for large cases is cccp + mincpp/parse - wc -2wc -3wc. Assumptions: cccp/mincpp are in same binary, mincpp read input from an in-memory buffer that cccp writes (cccp changed to write the buffer). Mincpp saves a read (-wc), cccp saves a write (-2wc), and mincpp does not write a Trace file (-3wc). Write I/O was measured as twice as slow as read I/O on the experimental system. The subtracting for no trace is justified since the trace file is a caching optimization that if turned on results in slightly higher times for the first run with the expectation of savings upon reuse. For the small cases we use cccp + mincpp/parse - 1wc as the numbers are too small to really matter.
100% effective replay caching	The build is needed, but the trace cache is 100% effective. While clearly overly optimistic, we will argue in Chapter 5 that 80-90% is a reasonable expectation. Preprocessing time is that for mincpp/replay.
100% effective caching, and build can be avoided.	As above, caching is effective, plus at the end the .min.i file is shown to be the same as the last one produced, and the build is short circuited. More detail is presented later on how this is done. Overall build time is that for mincpp/replay.
cccp time	Repeated here for spreadsheet calculation.
mincpp/parse mode	Repeated here for spreadsheet calculation.
cc time for .min.i file	Repeated here for spreadsheet calculation.

mincpp/replay mode	Repeated here for spreadsheet calculation.
wc time	Repeated here for spreadsheet calculation.

In comparing mincpp with standard cc, there are three cases to consider:

1. There is no replay, just parsing. cpp is still used for preprocessing. This is called the worst-case condition. In calculating this, we assume that cccp and mincpp are both modified to be in the same binary, and they use an internal in-memory buffer to reduce file I/O. Also the trace file is not written. The trace file is really a part of the caching strategy, and must be considered a first-time overhead cost to be repaid later via reuse.
2. There is 100% reuse of the trace cache, and the build is necessary. In other words, imagine that mincpp saves previous copies of .min.i files. It could then compare the present translation to a past one. If they are the same, it could just touch the .o file and bypass cc altogether. Otherwise, it calls cc so it can replace the .o. This case is for calling cc to create a new object file.
3. Same as 2 but it can just touch the .o.

Chapter 3 discussed replay mode and hinted at build avoidance. In Chapter 5 we will explore the issues of replay caching and build avoidance in much more detail.

Referring now to the chart, we see improvements that are meaningful even assuming the worst case of no caching. We also see that caching provides dramatic improvements over no caching. Build avoidance is almost an order of magnitude better. Overall, mincpp provides significant reductions in compile times for all three build scenarios.

Referring now to the table for the small cases, we see that mincpp worst case is about a draw with cc, that replay caching always helps some, and that build avoidance can help a lot. In any event, a reduction from 0.8 seconds to 0.6 seconds with 100% caching is not world-changing news. Build avoidance is still very helpful if a header file is changed. The basic conclusion is: if the program is a small C program then mincpp is not

going to be tremendously helpful unless a header file is changed. Despite this, there are some point development tools described in Chapter 5 that would be useful even in the small program environment.

4.7.7 Compiler Drivers

As discussed in the previous chapter, this work was originally conceived as an optimization to a C++ compiler system, not just for C alone. While the compile time improvements are significant just for C, as we have just seen, further improvements for C++ are anticipated. Taking advantage of the similarity between C and C++, one module from the benchmark suite was compiled with a variety of compilers ranging from a K&R C compiler to several commercially available C++ compilers. The results suggest that the improvements demonstrated here will increase with C++.

Table 4.9 Compiler Driver Statistics

tool	"-c"	"-g"	"-g".o size	repository
cc	2.1	2.4	45192	0
cc/mincpp/s	1	1.2	8508	195117
cc/mincpp/r	0.6	0.7	8508	195117
acc	2.8	3	55200	0
acc/mincpp/s	1.2	1.2	24824	195117
acc/mincpp/r	0.7	0.7	24824	195117
CC/hack .h	4.3	5.2	76556	0
CC/real .h	4.6	5.5	76552	0
CC/cback	5	5.3	8756	0
clc++	7.9	8.4	112048	0
clc++/ddcg	7	7.5	32116	0
clc++/ddcg/ch/s	30	30	32116	2500000
clc++/ddcg/ch/r	6	6.3	32116	2500000

- "-c": All compilers are compiling -c: non-optimized, non-debug.
- "-g": All compilers are compiling -g: full debug.
- "-g" .o size: Size in bytes of the object files produced by the full debug compiles.
- repository: Size in bytes of the compiled header file repository (in Centerline's case) or size of the trace file in bytes (in `mincpp`'s case). `Trace` file size is reported

uncompressed. Experiments with using text compression (`compress`) show 2-3 times reduction in file size over that reported.

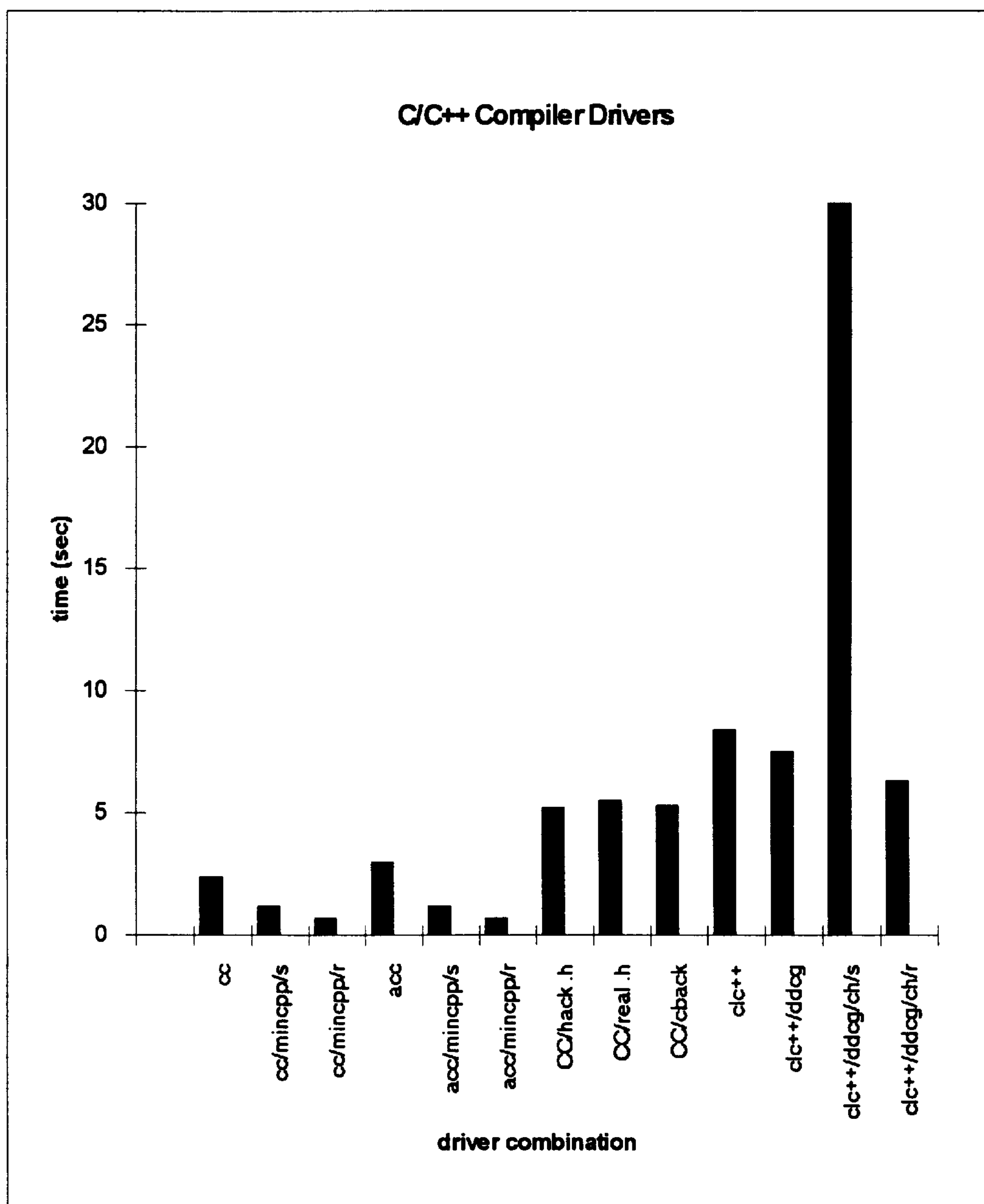

Figure 4.12 Compiler Drivers Compile Time

All times were measured using a Sun SparcStation 10/30 running SunOS 4.1.3 with a 1 gigabyte SCSI-2 disk and 128 megabytes memory.

Each row of the table corresponds to a different compiler system. The rows are described as follows:

1. cc, the Sun K & R C compiler.
2. cc, using mincpp in parse mode.
3. cc, using mincpp in replay mode.
4. acc, the Sun ANSI C compiler.
5. acc, using mincpp in parse mode.
6. acc, using mincpp in replay mode.
7. CC, the Sun C++ compiler, version 2.1. This is a cfront-based compiler. Function prototypes had to be added to the code since C++ requires a prototype and C does not. This is referred to as CC/hack.
8. CC, including the proper #include files to properly fix the compile error in 7.
9. CC/cback. CC using cback between cfront and ccom (ccom is the actual compiler binary under SunOS).
10. clc++, the ObjectCenter 2.0 compiler. This is a cfront 3.0 derived compiler. The compiler uses FlexLM to license compiles.
11. clc++ using demand driven code generation
12. clc++ using demand driven-code generation and compiled header files in the save case.
13. clc++ as in (12) but restoring the compiled header files in lieu of parsing.

What we see here is that the C++ compilers are decidedly slower than the C compilers. Since mincpp cannot (yet) parse C++ code, we can only speculate that the compile time improvements shown earlier in this chapter will be magnified for C++ code and provide even greater benefits. Another decided benefit is the efficiency of mincpp's "compiled header files" approach, which uses less disk space than the equivalent ObjectCenter repository binary but is more effective at reducing compile times.

4.8 Results Summary

Mincpp saves compile time by giving the compiler a smaller job to do. It takes less time for it to minimize with special purpose algorithms just for minimization than for the compiler to process the discarded code.

For small examples, there is little benefit to any optimization strategy, be it compiled header files, incremental compilation, or minimization. For larger applications, improvements in compile time and object file size are significant.

An important observation is that mincpp, on the examples chosen, never *increases* compile times. As we saw with the ObjectCenter example, that particular scheme seems to increase times dramatically in parse and save mode.

5. FUTURE WORK

5.1 Overview

This chapter is about what was not actually implemented for the present work but about the ultimate vision of the shape of a toolkit suitable for production use. At the beginning of Chapter 3 we discussed the implementation rationale for `mincpp`. In essence, what was implemented was designed to answer the basic question: can compile times be reduced by reducing the input to the compiler. The question was answered in Chapter 4. The goal of this chapter is to extend the core concept detailed previously and show its application in the larger context of the entire software development environment.

5.2 Mincpp System

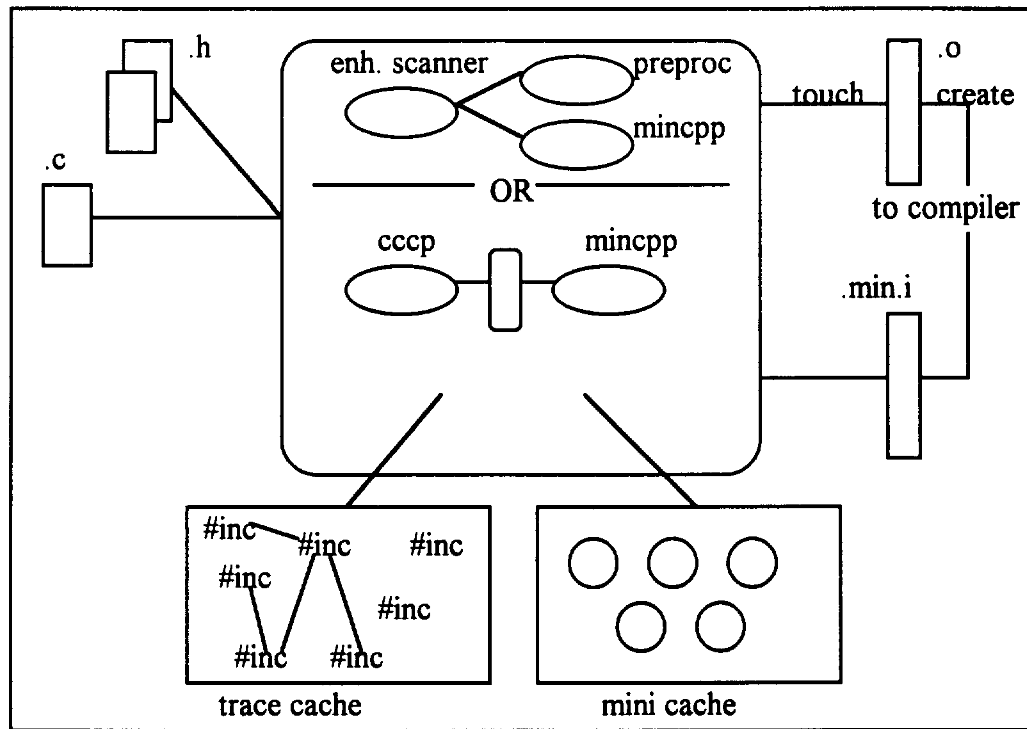

Figure 5.1 Mincpp System

Mincpp, as implemented, is the single oval labeled "mincpp" in the diagram above. Cccp is the GNU ANSI C preprocessor. The system reads source code as .c and .h files and writes a .min.i file. The box around the mincpp ovals will be called the "Hub" and provides all the glue code amongst the processing steps. The system uses two caches: one is a collection of previously created minimized .i files. The other is the "Trace Cache". This stores trace information in the same format as the trace file shown in Chapter 3, but it is indexed and arranged to permit a compact representation that suits the needs of multiple modules, multiple compiler switches, and multiple users.

5.2.1 Combined Parser

One of the glaring deficiencies of the work as implemented is the fact that mincpp needs to work with preprocessed code and does not actually replace the preprocessor. There is a large amount of wasted I/O that results. There are two solutions to this problem. One is very simple: compile the preprocessor code and mincpp code

into a single binary and let them share an in-memory buffer. Cpp writes this buffer with preprocessed text and mincpp minimizes it. This is easy to implement: some makefile work, finding the printf() statements in cccp and making them print to a string (sprintf()), and making mincpp's scanner read from this string, which just means changing its default read() procedure call to one that sets up a pointer to the buffer text.

Another single binary issue is the presence of two yacc grammars: one for preprocessing and one for parsing. Most yacc implementations have provisions for multi-parser binaries, but standard yacc does not. There is an easy workaround, however. We change the yy variables and functions in each generated parser to have a unique prefix using sed or awk with a filter such as 's/yy/pp_yy/g' and 's/yy/min_yy/g' to uniquely name the variables for the preprocessor and minimizer grammars respectively. The same issue, and solution, apply to the presence of two scanners in the same binary.

While this solution is straightforward, given a little effort to work around lex/yacc limitations, there is more we could do. Based on profiling results, we saw that 96% of mincpp's time was spent parsing, that is: time in generated lex and yacc code. We further observed that this time was evenly split between lex and yacc code. Scanning was just as expensive as parsing. Going one step further, we could enhance the lex code to include tokens for preprocessing tokens. This in essence means replacing the scanning code in cccp, which is non-lex based, with additional lex rules to match preprocessing directives such as #define, #undef, #if, and #include. The scanner would need to keep a list of defined macro's around to do text substitution, moving that code out of cccp and into the scanner. The #include macros would have the effect of pushing a new input stream in front of the scanner.

Probably the cleanest solution for the yacc files would be to keep them and their supporting C code separate. If the scanner finds a preprocessing token, it sends it to the cccp yacc parser, otherwise it sends it to the mincpp parser. Since yacc is normally the master of this process, calling yylex() when it wants to do look-ahead, we would have to modify the yacc control files (yaccpar in standard yacc or header/body/trailer

files in Berkeley yacc) to behave in a co-routine relationship with lex. This does not put lex in charge per say, but allows the two parsers to alternate execution based on the tokens encountered.

There has also been some recent work in creating a yacc-like tool that can deal with conditional parsing [Platoff94]. This could be an approach that would allow the two parsers proposed above to be consolidated into a single parser, with possibly additional savings in overhead.

5.2.2 Parse Avoidance and Trace Cache

Since mincpp spends the vast majority of its time parsing, and we want to make the execution time as small as possible, the obvious optimization is to avoid parsing.

We have shown how replay mode is significantly faster than parse mode. While the trace facility as demonstrated is not directly useful in a production compiler system, it forms a technical basis for a realistic scheme.

One idea is to treat the trace files as a tree of linked text that mirrors the structure of the include file hierarchy as shown graphically in Chapter 4. There is a link established that not only links the files, but also links the preprocessor options in effect when the link was made, especially the defined macros and their values, which can affect the contents of the preprocessed code. So, we could allow multiple copies of the link with different preprocessor options to be stored concurrently.

It would be most desirable to store the linked traces and the link context persistently. To do so, a database that allows objects and object relations to be directly expressed would be the most natural representation. This is not a surprise, since many design problems have been shown to be modeled most naturally with an OODB while many commercial business problems are modeled naturally using a relational model.

There are many OODB solutions at present that might be a good fit based on an analysis of their performance in the context of this application. ObjectStore, Ontos, Objectivity, Versant, and Raima [Raima92] are a few of the vendors that currently offer such products.

Another possibility would be to store the "indexed" traces in memory, but this would be very costly in terms of virtual memory consumed. Much better would be a caching scheme to keep hot pages in memory and the rest on disk, which is exactly what an OODB system does transparently.

Another issue is the invalidation of pieces of this cache. If a file's modification time is altered, we would choose to delete all trace information for that file.[21] While this may be more than needs to be deleted, it is surely easier to rebuild it than to figure out what changed. To make this really effective we would limit the trace cache to saving pieces of header files *only*, since the base C module will be modified more than the header files. We could also provide a directory filter so that the developer's private header files are also exempt from caching, and only external header files are cached. By using such a scheme, it should be easy to allow caching of infrequently changing files and specify parsing mode for frequently changing files.

In the system, the process would be started from the actions attached to the `#include` token processing. It would search for a matching cache element for the same file with the same preprocessor switches. If found it would load directly from trace and avoid parsing, otherwise it would parse and populate a trace tree for the run to be used later by other compilations.

It is asserted that most of the time the only files that are changing are tightly coupled header files and the C modules themselves. The majority of the compilation unit remains static: i.e. the foundation part of the program such as class libraries, windowing support (`windows.h` or `Xm.h` for example) or system-provided facilities. This assertion is what makes caching 80-90% effective.

---

[21] A list of pathnames for file components that have changed is available as a built-in `make` variable with GNU `make`. Additionally, if the command line to the compiler driver changes, invalidation must also occur since the order, number, or position of preprocessor options may have changed, which can alter the text of the preprocessed code.

5.2.3 Build Avoidance and `.min.i` Cache

Every programmer has experienced the problem of changing a header file and waiting a long time for a build to complete while `make` rebuilds the entire application. While the trace cache would make this process much faster than usual, we introduce the `.min.i` or "mini" cache as a further optimization.

If `make` decided that, based on file time-stamps, a build is necessary, but after minimization no text difference from the last build for the module is detected, `mincpp` could touch the object file and avoid compiling altogether. In the development of `mincpp` itself, a common case was adding a function to the symbol table on behalf of a single module elsewhere, and having to recompile the entire application unnecessarily. As a concrete example, function `hook_text()` was added for a routine in `r26.c`. Only `symbol.c` that implements the function, and `r26.c` that references the function really need to be compiled, but since `symbol.h` is included everywhere the whole application is rebuilt by `make`.

To do this, verbatim copies of previously created "mini" files are kept in a directory, tagged with the preprocessor options that were in effect, and possibly the modification time-stamps of all the constituent files. When a new `.min.i` file is created by `mincpp`, it could search the "mini" cache for likely matches by comparing file size and preprocessor switches. Using `diff` or `sum`, matches can be detected.

If header files are stable and unchanging, then the "mini" cache could be disabled. If header files are mostly static but are sometimes changed, with long builds resulting, the cache can be used to reduce the cost of header file changes.

5.2.4 The Hub

The agent that coordinates the activities of the trace cache manager, "mini" cache manager, and consolidated parser is collectively called the Hub. This code would scan for matches in the caches, manage the state transitions between parse/replay, invalidate cache entries, prune caches to meet size constraints, and interact with `make`. It would also

replace `cc` as compiler driver since to effectively touch object files it needs to be in control of the compiler driver.

5.2.5 Multi-Threaded Compile Server

A compile server could run on a number of machines on a network. The server would accept a compilation job by `make`, possibly using a network-distributed version of `make`. The server would read the job out of `make's` environment, taking the command line, environment variables, and `make` built-in variables. It would attach the job to a thread waiting in a ready thread farm and let the thread perform the work. The server is then free to accept a new job while the thread is completing the job. The thread would open and lock entries in the caches in a manner consistent with multi-user access and guarantee freedom from deadlock. In particular, the trace cache needs to be locked as it is descended to keep other readers/writers from locking ahead, which could result in a race condition that could lead to deadlock.

Use of threads would also enable parallel execution of compiles in environments that support such on a multiprocessor, such as Solaris 2.x, OSF/1, and Windows NT. A server implementation would also more effectively use any disk caching mechanism provided by the database.

5.3 Ancillary Tools

So far, we have focused exclusively on `mincpp`'s application in the compiler driver itself with the goal being to reduce compile times. There are a number of other applications of the basic technology that merit some discussion.

5.3.1 Minimal `#include` Generator

The compiler forces the programmer to include a new header file when a reference to a new file is made in the code. But, when the code changes in a way that allows an included file to be deleted, it is most likely that the included file will remain included. Most programmers are too busy to tinker with compiling, working code. A simple tool that peruses the minimized output could generate a minimal `#include` block for the module.

An even more useful tool could be developed by just adding a global index to the trace cache. If a new function or type is referenced by new code, this facility could assist the programmer in identifying the correct `#include` and place it in the code automatically. We could add a regular expression lookup for incomplete lookups. Currently, the programmer identifies a need to use a function, usually fumbles through `man` pages to hunt it down, maybe cracks a book or two, finds it, has to move up in the file to add the `#include`, pages back to where they originally were, and then proceeds.

Using this tool, the function could be queried, a choice box displayed with matches and their corresponding `#include` files, and indication of what `#include` files are already in the module that could bind the function. A simple selection and OK replace the above-mentioned tedious procedure.

5.3.2 "Presto" Syntax Checker

`Mincpp` could be integrated in an editing environment such as `emacs` [GNU87] through the mock Lisp interface. This would allow the code to be run through `mincpp` in a quick and interactive fashion. This would be a way to do a quick syntax check without incurring the overhead of doing a full compile. On a syntax error resulting from a missing header file, selecting the text and invoking the lookup facility described in the last section would be a way to quickly add the necessary `#include` file. It would be possible to run an automatic syntax check in the background and display an unobtrusive indicator on the display: red for syntax error, green for OK.[22] If the programmer thinks that the code should be correct but sees red, it is an immediate indicator to look for a problem before the problematic code has been cut and pasted somewhere else, compounding the problem.

5.3.3 "Dead" Text Eliminator

`Mincpp` is based on a module by module view of the compilation process. It would be possible to add a subassembly view based on a collection of modules that constituted a program or library, and above that an assembly, or system, view that

---

[22]Remember that since `mincpp` is based on a full C parser it also detects all syntax errors.

represents a body of applications that represent a complete software system. Once this is done some analysis on the whole can be done based on minimization. Dead functions and definitions could be identified as candidates for deletion, based on the judgment of the developer. Just as is the case for stray, unnecessary `#include` files at the module level, dead declarations, type definitions, and functions rarely are cleaned up from a compiling, working system. A tool such as this could be used at key points in the development cycle to give the code a quick tune-up before going into production.

5.3.4 File Architect

Minimization results could drive graphical tools that examine the relationships and dependencies between modules and their header files. This information might lead to a re-organization of header files based on reference patterns. For example, a header file might be split into two: one for specialized facilities used by just a couple of modules, the other being used for the more commonly used facilities that are of more general interest.

Metrics could be developed to aid in this procedure and identify possible candidates for examination. Minimization results provide insight into the code that is different from that available to either static or dynamic analysis tools.

5.3.5 Dependency Manager

`Mincpp` could interact with `make` in a tightly coupled fashion. `Mincpp` could maintain the file dependency lists for each module and update it on every compile, ensuring "correct by construction" dependency lists. This prevents dependency leaks from occurring. If dependencies are not correct by construction there exists the potential for a build to be missed by `make`. For a `make` that supports conditional `makefiles`, or an environment that uses preprocessed `makefiles`, `mincpp` could generate "`#define` smart" dependency lists, that handle conditional inclusion, a very common portability technique.[23] This is actually how the stated problem of "too few builds" cited in Chapter 1 is addressed.

---

[23] `#ifdef WINDOWS #include <windows.h> #else #include <Xm.h> #endif`

Other interactions with make include reading the environment out of make as described in the compile server description, and the touching of object files as a control mechanism that the "mini" cache can enable. There are proprietary build environments that might also allow a custom integration of mincpp technology through a vendor API.[24]

5.4 C++

Mincpp's minimization algorithms could be extended to work with C++. The C parser that mincpp uses is actually a subset of a C++ parser also supplied with the package. While this parser does not handle the C++ 3.0 features of templates and exceptions, it does handle everything else.

The additions would involve treating class derivation as type dependency, treating functions as complete signatures to handle function overloading, and different name space treatment [AT&T90].

Also, since the bulk of mincpp's functionality does not depend on the reference yacc parser in particular, it could be easily re-hosted to a compiler environment that uses a custom parser, or a parser that uses different compiler-compiler tools.

Re-hosting into a custom compiler environment would eliminate the overhead of creating the intermediate .i file, but only at the expense of reduced portability.

5.5 Aggressive Minimizations

It should be noted that mincpp's algorithms could be made smarter and perform more minimization than is presented here. Such a case is "pointer references" to structures where an abstract declaration can be substituted for the full declaration if the only references are by pointer. In C++ this would also include "by reference".

In addition to this, unused local variables could be eliminated. Further, the notion of reference to structure could be significantly refined to really be "reference to structure member". This opens up several intriguing possibilities such as only disclosing members up to the last lexically referenced one (so the compiler gets sizeof right...), or disclosing

---

[24] Atria's ClearMake [Atria93], part of the ClearCase product, is a good example.

only referenced members, substituting "slop" members for correct space and alignment. The latter obviously is platform and compiler switch dependent. Also, careful treatment of virtual functions in a C++ system would have to be considered.

5.6 Side Effect Management

5.6.1 Compiler/Debugger

`Mincpp` as currently implemented reports incorrect line numbers that end up being reported in compiler warning and error messages, and subsequently in source listings available under a source-scrolling debugger. There are very straightforward techniques available to keep track of line/file numbers in `lex`. `Mincpp` should store line ranges and file names along with the text fragments it indexes. Then, if a text fragment or series of text fragments are skipped, the next fragment whose text is emitted would reset the file and line number by emitting a `#line` directive, keeping track of the number of skipped lines by adding to a total from a line-count member in the text structure. This is the approach is used in [Litman93].

The symbols for all those fragments that are eliminated are not available for use in the debugger, but this should not be of concern to the programmer. If a structure is not used in a module, there is little chance that a programmer would want to view an object casted to such a structure anyway.

5.6.2 ANSI "Implementation-Defined Behavior"

There are opportunities for problems in code that takes advantage of implementation-defined behavior. Suppose there are three declarations for objects that are lexically sequential such as `int i; int j; int k;`. Furthermore, the programmer has taken the address of `i` such as `void * p = &i;`. The programmer knows the program is running on a Sun SparcStation, and `int`s are 4 bytes, so are `void *`'s, and the programmer accesses `k` by adding 8 bytes to `p` (`p += 8; *p = 4;`). If `k` is not explicitly referenced, it's gone from the text.

This is poor coding practice and is certainly not portable. One option is for `mincpp` to not discard space allocating declarations, only type definition or function definitions. The other option is to support a `#pragma` directive to allow the programmer to explicitly state that the following declaration is not a candidate for minimization. For example:

```
int i;
#pragma keep
int j;
#pragma keep
int k;
```

This also applies to objects stored in a module that are not referenced, but have some external use:

```
#pragma keep
char *SCCSId = "version 1.2 file.c copyright (c) 1994 \
          Etherium Software";
```

The general problem of space allocating declarations is treated in more detail in the next section.

5.6.3 `Extern` Objects

If a global variable is declared but not used in file A and file B uses the variable via an `extern` declaration, `mincpp`, as implemented, will erroneously remove the declaration of the variable from file A. The code in file B is counting on the space for the variable to be created for it somewhere.

Some compiler systems allow space allocating variables in header files. The linker resolves all references to a same-named global variable to the same instance. Other compilers do not allow this but force space allocating objects to be declared in a header file via an `extern` declaration, and the programmer must allocate the space for the variable once in a single C module. Certainly, the latter style is more portable and depends less on the linker implementation. Many ANSI C and C++ compilers do in fact enforce the latter behavior.

This problem was not encountered using the sample benchmarks used in Chapter 4.[25] So, while this effect was not observed here, there are several approaches to remedy the situation that could be implemented.

1. Do not minimize away space allocating declarations. This can be easily implemented by setting the reference bit for a declaration if the non-terminal **declarator** is recognized at any point during the parsing of the declaration. This would effectively keep such declarations just as function definitions are kept. This could possibly reduce the amount of minimization, but in well-structured programs the number of global variables is small relative to everything else, and as described above, space allocating declarations are rare in header files for portability reasons.
2. Provide a `#pragma` directive to direct `mincpp` to not allow the following declaration to be minimized away. This is an inferior solution that would require source code changes.
3. Keep a list of space allocating declarations that have been minimized away internal to the tool. After the first link, read the undeclared symbol list from the linker and generate a C module that is the text for the symbols that were minimized away. Compile this, add it to the list of modules for the linker to process, and link a second time with the minimization stub now resolving the undefined references. This is also an inferior solution as it requires a two-pass link.
4. During the printing of minimized text, the tool creates a library entry for each space allocating declaration that is skipped. This is done by creating a C module for each discarded declaration, compiling it, and adding it as a library member. This library is added at the end of the link list. This approach is the preferred solution: it makes compile times as small as possible, it only requires a one pass link, and any space-allocating global declarations that really are not referenced anywhere are omitted from the binary, saving space in the program's data section. There is a small amount of overhead in maintaining the library. This approach has an additional benefit for C++

---

[25]If this were to be a problem for the benchmarks, these variables would have produced undefined symbols during the link. Since this was not the case, the minimization results remain valid.

systems. In C++ a static object instance in a header file or elsewhere can result in excess static constructors being called at program start up and excess static destructors being called at program exit. Standard header files for streams and complex numbers contain such static objects. If these header files are pulled into every module, as is often the case, then invoke times can be degraded while the extra objects are being created. The library approach collapses all of these object instances into just one object. In a traditional `cfront/ld` environment, each module has a *separate* copy of every static object mentioned in the preprocessor output.

5.6.4 Syntax/Semantic Compiler Errors

As observed in [Litman93], another side effect of including a parser in the preprocessor is that syntax errors are reported during preprocessing, not later during compilation. The users of [Litman93] reported that having syntax errors reported first, with semantic errors and type checking errors reported later was actually beneficial and clearer.

6. CONCLUSION

6.1 Key Benefits Demonstrated

Despite the limited nature of the system actually implemented, we were able to demonstrate several key benefits of using `mincpp` in a compile environment.

6.1.1 Reduced Compile Times

`Mincpp` reduces compile times by reducing the quantity of input given to the compiler. In many applications, the volume of text added to each module in the form of header files is much greater than the volume of code. All that extra text takes time to process. If it can be minimized by a special purpose tool, savings in overall compile times can be demonstrated. For C programs, we saw that the savings were modest for smaller examples, and much more significant for larger application that use "heavy" header files, such as GUI applications based on the X windows system.

We also saw that the complexity of the grammar the compiler processes can increase this benefit. For one example, the savings increased dramatically as the compiler for the same module was changed from K&R C, ANSI C, C++ 2.1, and C++ 3.0.

6.1.2 Compiler, Operating System, and Platform Independence

A distinguishing characteristic of `mincpp` is its ability to improve compile time no matter the compiler, operating system, or hardware platform used. It can be deployed with any system without `makefile` or source code changes. It does not introduce any new "rules of engagement" that the programmer is forced to carry around in their head. Such rules are inevitably forgotten, broken, and ignored.

6.1.3 Reduces Object File Sizes

All of the text fragments omitted take up space in the object file's symbol table. This space is saved by using `mincpp`. The same space saved in the object file is potentially saved in the executable file if a particular text fragment is completely dead. That is, it is not referenced by any constituent module of those that make up the complete binary.

6.1.4 Build Avoidance and `make` Dependency Leaks

Changing a header file can cause a complete rebuild, since the tools used to decide what has to be rebuilt can only check file granularity changes via file time-stamps. This is very pessimistic, but at least produces something less than a complete rebuild in most cases. This problem can be solved by letting `mincpp` short circuit re-compiles if it can determine that a requested build is not necessary, since the minimized text is identical to the prior materialization of that particular module.

There are more sophisticated builders that allow object files to be shared amongst developers, such as Atria ClearCase. These systems allow one developer to rebuild the system, check in source code changes, and the configuration management system also "checks in" the matching object files. Subsequent builds then use these object files since they match. Unfortunately, they do nothing to help the unfortunate developer who had to rebuild the entire application in the first place.

Despite all this, a dependency leak in a `makefile` can introduce a difficult to find run-time problem if something that should have been built was not. `Mincpp` can generate file dependencies that are correct by construction and guarantee that this problem will not occur.

6.2 Key Benefits Suggested

There are some specific benefits that were alluded to but perhaps not explicitly stated in Chapter 5. We state a few of these here.

6.2.1 C++ Static Initialization

In C++, a static object instance that has a constructor has its constructor called before `main()` is called. A static object instance that has a destructor has its destructor called after `main()` exits. If a program has many of these, it can have an impact on the time from program request until the user can interact with the program, called invoke time, and the time from the user exiting the program until it is gone, called tear-down time. With enhancements proposed in Chapter 5, the extra objects are eliminated.

6.2.2 Network Traffic

A compile server such as described in Chapter 5 can have a very positive effect on network traffic in a multi-node parallel build environment. Since the server is mostly running out of local, indexed, memory-paged trace cache, it is not reading files over a network-oriented file system. Not only is this much faster, but by reducing network traffic it allows the network to support more concurrent operations/nodes than it would be able to support otherwise.

6.2.3 Point Development Tools

`Mincpp` has some incremental benefits even in the edit cycle, if some tools based on its minimization technology are developed.

The include generator reduces the amount of time managing include lists. It also reduces the time spent hunting for definitions.

The syntax checker allows a quick check for syntax errors to be found and fixed before doing a full compile. When the programmer does a build, they do so with some assurance that syntax errors will not be reported.

6.2.4 Architecture and Metric Tools

For header files that the developer has some control over, `mincpp` can help the programmer manage the partitioning of these files. `Mincpp` can provide reference metrics and some sort of "min-cut" analysis of the use of files across an application. This may result in a reorganization of files if it makes sense.

6.3 Closing Remarks

Mincpp has been (partially) implemented and characterized. We have seen compile time improvements even in its worst case mode, and substantial improvements using caching. Mincpp offers several benefits that could result in increased developer productivity and satisfaction.

References

[ANSI88] American National Standards Institute, *C Language Specification*, American National Standards Institute, New York, New York, 1989.

[Aho86] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.

[Atria93] Atria Software, Inc., *ClearCase Reference Manual*, Atria Software, Inc., Natick, Massachusetts, 1993.

[AT&T90] AT&T, *Unix System V AT&T C++ Language System Release 2.1 Selected Readings*, AT&T, Murray Hills, New Jersey, 1990.

[Borland93] Borland International, Inc., *Borland C++ User's Guide, Version 4.0*, Borland International, Inc., Scotts Valley, California, 1993.

[Centerline93] Centerline Software, Inc., *ObjectCenter Reference Manual*, Centerline Software, Inc., Cambridge, Massachusetts, 1993.

[CenterLine93a] Centerline Software, Inc., *TestCenter Reference Manual*, Centerline Software, Inc., Cambridge, Massachusetts, 1993.

[Ellis90] Ellis, Margaret A., and Stroustrup, Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[Feldman86] Feldman, S. I., "Make - A Program for Maintaining Computer Programs", PS1:12-1 - PS1:12-9, BSD4.3 Unix Programmer's Manual Supplementary Documents 1, University of California, Berkeley, California, 1986.

[Harbison91] Harbison, Samual P., and Steele, Guy L. Jr., *C: A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Gircys88] Gircys, Gintaras R., *Understanding and Using COFF*, O'Reily & Associates, Inc., Newton, Massachusetts, 1988.

[GNU87] Free Software Foundation, *GNU Emacs User's Manual*, Free Software Foundation, Cambridge, Massachusetts, 1987.

[GNU92] Free Software Foundation, *GNU Make User's Manual*, Free Software Foundation, Cambridge, Massachusetts, 1992.

[Holub93] Holub, Allen, "Visual C++: It's Compiler, Language Implementation, and Code Quality", pp. 65-73, Microsoft Systems Journal, Vol. 8 No. 6, June 1993.

[Iitsuka93] Iitsuka, Takayoshi, and Kikuchi, Sumio, "Method of Recompiling a Program by Using Result of Previous Compilation", United States Patent 5,230,050, July 20, 1993.

[Jaeschke89] Jaeschke, Rex, *Portability and the C Language*, Hayden Books, Indianapolis, Indiana, 1989.

[Johnson86a] Johnson, S. C., "A Tour Through the Portable C Compiler", SMM:19-1 - SMM19-28, BSD4.3 Unix System Manager's Manual, University of California, Berkeley, California, 1986.

[Johnson86b] Johnson, Stephen C., "Yacc: Yet Another Compiler-Compiler", PS1:15-1 - PS1:15-33, BSD4.3 Unix Programmer's Manual Supplementary Documents 1, University of California, Berkeley, California, 1986.

[Kernighan88] Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Kernighan84] Kernighan, Brian W., and Pike, Rob, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Knuth73] Knuth, Donald E., *The Art of Computer Programming: Volume 3 / Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973.

[Lesk86] Lesk, M. E., and Schmidt, E., "Lex - A Lexical Analyzer Generator", PS1:16-1 - PS1:16-13, BSD4.3 Unix Programmer's Manual Supplementary Documents 1, University of California, Berkeley, California, 1986.

[Lucid93a] Lucid, Inc., *Lucid C++*, Lucid, Inc., Menlo Park, California, 1993.

[Lucid93b] Lucid, Inc., *Energize Programming System User's Guide*, Lucid, Inc., Menlo Park, California, 1993.

[Levine92] Levine, John R., Mason, Tony, and Brown, Doug, *Lex & Yacc*, O'Reilly & Associates, Inc., Sabastopol, California, 1992.

[Lippman91] Lippman, Stanley B., *C++ Primer*, Addison-Wesley, Reading, Massachusetts, 1991.

[Litman93] Litman, Andy, "An Implementation of Precompiled Headers", pp. 341 - 350, Software-Practice and Experience, Vol. 23(3), March 1993.

[McKeeman92] McKeeman, William M., and Aki, Shota, "Incremental-Scanning Compiler for Source-code Development System", United States Patent 5,170,465, December 8, 1992.

[Microsoft91] Microsoft Corporation, *Microsoft C/C++: Environment and Tools*, Microsoft Corporation, 1991.

[Onodera93] Onodera, Tamiya, "Reducing Compilation Time by a Compilation Server", pp. 477-485, Software-Practice and Experience, Vol. 23(5), May 1993.

[Platoff94] Platoff, Michael, and Wagner, Michael, "Method for Parsing and Representing Multi-Versioned Computer Programs, for Simultaneous and Synchronous Processing of the Plural Phases", United States Patent 5,276,880, January 4, 1994.

[Pure93a] PURE Software, Inc., *Purify User's Manual*, Pure Software, Inc., Sunnyvale, California, 1993.

[Pure93b] PURE Software, Inc., *Quantify User's Manual*, Pure Software, Inc., Sunnyvale, California, 1993.

[Raima92] Raima Corporation, *Raima Object Manager Reference Manual*, Raima Corporation, Issaquah, Washington, 1993.

[Sedgewick90] Sedgewick, Robert, *Algorithms in C*, Addison-Wesley, Reading, Massachusetts, 1990.

[Smith93] Smith, Steven P., Padawer, Andrew D., Jones, David T., Whitten, Gregory F., and Wittenburg, Craig H., "Incremental Compiler", United States Patent 5,204,960, April 20, 1993.

[Stroustrup91] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1991.

[Tuthill86] Tuthill, Bill, and Dunlap, Kevin J., "Debugging with dbx", PS1:11-1 - PS1:11-6, BSD4.3 Unix Programmer's Manual Supplementary Documents 1, University of California, Berkeley, California, 1986.

Biographical Note

The author was born 18 September 1962 in Spokane, Washington. He attended St. Charles Elementary School from 1968 and graduated in 1976. He attended Gonzaga Preparatory School, graduating Summa cum laude in 1980. His undergraduate degree is in Computer Engineering and was granted by Boston University, Summa cum laude, in 1984. The author pursued an active role in the Boston University's School for the Arts with a piano performance focus during this time. The author began his professional experience with Texas Instruments in Lewisville, Texas, but discovered a short time later that military electronics was not his calling. Between late 1984 and 1986 the author worked at Digital Equipment in Littleton, Massachusetts, on a VAX research project that later provided corporate memory for the development of parallel VAX and ALPHA processors, and symmetric multiprocessing in VMS. In 1986 the author transferred to Digital Equipment's R&D facility in Bellevue, Washington, otherwise known as DECwest. Here the author was a member of the team whose work later became the ALPHA project at Digital and the Windows/NT project at Microsoft. Late in 1989 the author left Digital eager to work on a project that "made it out the door", and assumed a key role in the development of release 8.0 of IDEAstation at Mentor Graphics Corporation. While in this role author was motivated to investigate the material that constitutes this thesis. In May 1992 the author assumed his current role of project leader and technical architect of a family of Clinical Information System products at SpaceLabs Medical in Redmond, Washington. While at DEC/Littleton, the author spent a year in the M.S.E.E. program at Tufts University, Medford, Massachusetts. While at DEC/Bellevue, the author took graduate courses in the C.S. department of the University of Washington as a Graduate Non-matriculated student. While at Mentor Graphics, the author was accepted into the doctoral program at Oregon State and passed the qualification exam before transferring to the Graduate Institute to

finish (finally) his Master's program. The author has been married six years to the former Julie Rassmussen and has two children: Madeleine (age three) and Peter (age one).