A Caching Compiler for C

by

Brian Keith Koehler
B Math , University of Waterloo, 1992

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

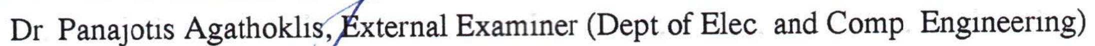
in the Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr R Nigel Horspool, Supervisor (Dept of Computer Science)

Dr Hausi A Müller, Departmental Member (Dept of Computer Science)

Dr Qiang Wang, Outside Member (Dept of Elec and Comp Engineering)



Dr Panajotis Agathoklis, External Examiner (Dept of Elec and Comp Engineering)

© BRIAN KEITH KOEHLER, 1994

University of Victoria

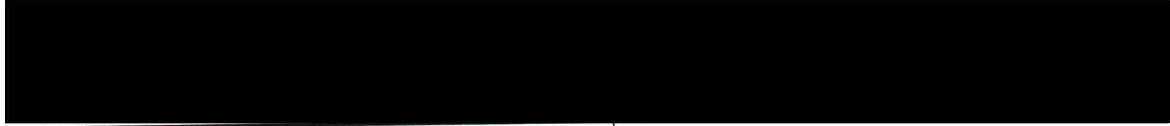
All rights reserved Thesis may not be reproduced in whole or in part, by photocopy or
other means without permission of the author

Supervisor: Dr. R. Nigel Horspool

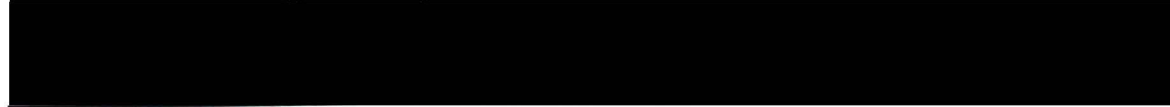
ABSTRACT

Conventional C compilers tend to spend a great deal of time processing text contained in header files. Worse yet, the same text may be processed over and over. This occurs when a programmer is either engaged in the repetitive edit-compile-debug cycle or when a massive compilation is performed and the same header files are repeatedly included by several compilation units. One approach to this problem has been to design compilers which use precompiled versions of header files that can be processed much more rapidly than the original source text of the header files. The primary drawback of this strategy is that the meaning of a header file is contextually dependent on where it is included. Since precompiled headers must be generated in advance then, in instances where the current compilation context does not match the context in which the header was precompiled, the source text of the header file must be reprocessed. We present a more general, transparent, adaptive scheme whereby internal representations of header files are generated and reused as actual compilations are carried out. The principal benefit is that context information is taken directly from real compilations and if the context changes then new internal representations can be generated automatically. Our scheme performs at least as well as implementations using precompiled headers, but is more general and completely transparent to the user. It achieves savings of 60% on average in the repetitive case and 40% in the massive case over the conventional compiler from which the compilation server was derived. The improvement comes at the expense of some increased use of memory resources.

Examiners

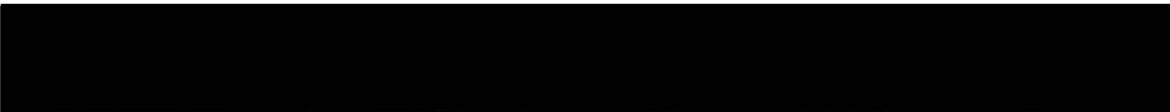


Dr. R. Nigel Horspool, Supervisor (Department of Computer Science)



Dr. Hausi A. Müller, Departmental Member (Department of Computer Science)

Dr. Qiang Wang, Outside Member (Dept. of Elec. and Comp. Engineering)



Dr. Panajotis Agathoklis, External Examiner (Dept. of Elec. and Comp. Engineering)

Table of Contents

ABSTRACT ii
Table of Contents iv
List of Tables vi
List of Figures vii
Acknowledgment viii
1 Introduction 1
1.1 Overview of compilation 3
1.2 Separate compilation 5
1.3 The problem 7
1.4 Thesis overview 9
2 Background 10
2.1 Eliminating redundant compilation of modules 11
2.1.1 Cascading recompilation 12
2.1.2 Smart recompilation 13
2.2 Eliminating redundant compilation within modules 17
2.2.1 Precompiled headers 17
2.2.2 Compilation servers 20
2.3 Summary 22
3 Approach 24
3.1 Equivalence of C program fragments 25
3.1.1 Compilation state 26
3.1.2 Preprocessing state 26
3.1.3 The state of the compilation proper 27
3.2 Identifying equivalent fragments 28
3.3 Practical limitations on testing for equivalence 35
3.3.1 Equivalent preprocessing states 35
3.3.2 Equivalent compilation states 41
4 Implementation 42
4.1 Overview 43
4.2 A preprocessing server based on cpp 44
4.2.1 Preprocessing state representation 45
4.2.2 Internal representation of header files 46
4.2.3 Identifying consistent states and updating state 47
4.2.4 Memory management 47
4.3 A compilation server based on lcc 49
4.3.1 Compilation state representation 50
4.3.2 Internal representation of header files 50
4.4 Reusing IRs in the compiler proper 51
4.4.1 Deferred semantic analysis of declarations - laziness 53

4 4 2 Memory management 57
5 Results 59
5 1 Declaration usage 59
5 2 Methodology 61
5 3 Measurements 62
5 3 1 Initial compilation 63
5 3 2 Repetitive compilation 66
5 3 3 Massive compilation 68
6 Conclusions and future work 72
6 1 A faster C compiler 72
6 2 Future work 73
6 2 1 C++ 74
6 2 2 Ada 74
6 2 3 Modula-2 77
References 80

List of Tables

1 Relative performance of preprocessors 45
2 Relative performance of combined preprocessors/compilers 49
3 Data segment sizes 66
4 IR usage 71

List of Figures

1 Trivial program 2
2 Monolithic compilation 5
3 Separate compilation 6
4 Program fragment with dependencies 11
5 Dependency graph for program of Figure 4 12
6 Litman's precompiled headers 18
7 COB symbol table 21
8 Structure of conventional compilation 25
9 Relationship between source text, target text, and compilation state 33
10 Refined version of Figure 9 34
11 Original compiler organization 43
12 New compiler organization 44
13 Example use of an IR 48
14 Reuse of compiler IRs 52
15 Type equality example fragment 54
16 Incomplete type example #1 55
17 Incomplete type example #2 56
18 Lazy order example 57
19 RUV distribution for macro identifiers 61
20 RUV distribution for all other identifiers 62
21 Distribution of relative initial preprocessing times 63
22 Distribution of relative initial compilation times 64
23 Distribution of relative initial overall times 64
24 Relative increases in preprocessor data segment size 65
25 Relative increases in compiler data segment size 65
26 Distribution of relative repetitive preprocessing times 67
27 Distribution of relative repetitive compilation times 67
28 Distribution of relative repetitive overall times 68
29 Distribution of relative massive preprocessing times 69
30 Distribution of relative massive compilation times 69
31 Distribution of relative massive overall times 70
32 Ada example 75
33 Modula-2 example 77

Acknowledgment

I would like to thank my supervisor, Dr Nigel Horspool, for his guidance and advice during the course of this research as well as his financial support in the form of a research assistantship without which this work would not have been possible I would also like to thank the rest of my committee for their help as well as my "unofficial" committee member, Jim Uhl, who has been a constant source of insight and constructive criticism

1 Introduction

Imagine that a book, *How to Achieve Financial Independence Through Widgets*, is a national best-seller. The author quickly realizes that having a Canadian national best-seller is no way to achieve financial independence. In order to capitalize on the lucrative American market she decides to have her book translated and published in American English. Having never employed the services of a translator before, she pays close attention to the work of her Canadian-American translator. She is stunned when he spends several days reading large tracts of the dictionary as if he were completely unfamiliar with the Canadian language. Finally, he begins work on Chapter 1 and she relaxes as he polishes it off in a few hours. Unfortunately her sense of relief is apparently premature as he again spends several days with the dictionary before even looking at Chapter 2 which, once he gets started, is also finished in a matter of hours. This painfully long process is repeated for each of the remaining chapters of her book leaving the author relieved that the task has been completed yet suspicious as to whether it could have been finished faster.

This seemingly intolerable scenario is remarkably similar to the situation programmers routinely encounter, possibly dozens of times a day, when they make use of a compiler. The only thing that makes the process seem bearable to programmers is that the absolute amount of time spent compiling a program or its parts is typically on the order of seconds or minutes rather than days or weeks which our imaginary translator might be spent translating a book. Yet, like our fictitious writer, the programmer has an intuitive sense that the

compilation process frequently takes longer than really seems necessary. As a concrete example, consider the small program fragment in Figure 1. This verbose program creates a

```
#include <X11/Intrinsic.h>
#include <X11/Shell.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Form.h>
#include <X11/StringDefs.h>

XtAppContext        Xtac;
Display             *dpy;
static Arg          args[20];
void                exit();

void main(argc, argv )
int argc;
char *argv[];
{
    Widget shell, form, button;

    XtToolkitInitialize();
    Xtac = XtCreateApplicationContext();
    dpy = XtOpenDisplay( Xtac, NULL, "hello", "Hello", NULL, 0, &argc,
            argv );
    shell = XtAppCreateShell( "hello", "Hello",
            applicationShellWidgetClass,dpy, NULL, 0 );
    form = XtCreateManagedWidget( "form", formWidgetClass, shell,
            args, 0 );
    XtSetArg( args[0], XtNlabel, "Quit" );
    button = XtCreateManagedWidget( "quitButton", commandWidgetClass,
            form, args, 1 );
    XtAddCallback( button, XtNcallback, exit, 0 );
    XtSetMappedWhenManaged( shell, True );
    XtRealizeWidget( shell );
    XtAppMainLoop( Xtac );
}
```

Figure 1. Trivial program

window with a button labelled 'Quit'. When the user clicks on the button the program exits. The unsuspecting programmer might be surprised to discover that this tiny 851 character program, which generates a mere 2,675 characters of assembly code, takes a whopping four seconds to compile (on a SPARCstation IPC). Assuming the machine executes around ten million instructions per second, the implication is that the compiler executes approximately 11,000 instructions per character read or written which seems horribly inef-

ficient. What is more infuriating is that even after a simple change to the program, such as changing the label of the button from ‘Quit’ to ‘Exit’, the program still takes four seconds to compile. It is not the length of a single compilation which is most frustrating but the cumulative effect of repeatedly waiting for the compiler after making a small change to the program, which is characteristic of the edit-compile-debug cycle programmers frequently find themselves locked into.

A possible solution to this problem would be for programmers to adopt a more Zen-like attitude to their work in which case compilation time would be irrelevant. However we acknowledge that it is generally easier to speed up the technology than reduce the expectations of society. Recent history is replete with inventions reflecting contemporary western culture’s growing impatience: touch tone dialing, microwave ovens, “instant” teller machines, one hour photo finishing, and drive through restaurants. As a result, this thesis examines a technique for making the compilation of C programs more efficient in the context of the repetitive edit-compile-debug cycle in an attempt to relieve the anxiety felt by programmers everywhere. The remainder of this chapter elaborates on why conventional C compilers are so inefficient in this regard.

1.1 Overview of compilation

A conventional compiler is a specialized form of translator which transforms high level descriptions of algorithms written by and/or intended to be read by people into low level sequences of instructions intended to be executed by a particular hardware platform. Some desired characteristics of a compiler include: correct code generation, efficient code generation, efficient generated code, effective error recovery and informative diagnostics.

The relative importance of each of these characteristics in a particular compiler depends on the intended use of the compiler. In a teaching environment, informative diagnostics and good error recovery are desirable. In a production environment, the efficiency of the generated code is extremely important.

In any implementation, correct code generation is essential. The semantics of the source text must be equivalent to the semantics of the generated target code. At best, incorrect code generation makes the source text of a program extremely difficult to debug and, furthermore, it may be tedious to find equivalent source text for which correct code is generated; at worst, there may be no equivalent source text which generates the correct target code.

Informative diagnostics help a programmer quickly locate lexical or syntactic errors in the program text. An informative diagnostic would at least include information regarding the exact or approximate location of the error such as file name, line number and column number, and also describe the nature of the error (for example, unrecognizable lexical element, redeclaration of an identifier, use of an undeclared identifier, missing statement separator or terminator, or unbalanced parentheses in an expression). Error recovery refers to the ability of a compiler to continue processing the remaining source text after an error is encountered. Combined with informative diagnostics, good error recovery should allow as many errors as possible to be identified in a compilation unit with a single run of the compiler.

The efficiency of generated code can be measured on two scales: time and space. Time efficiency refers to the execution speed of the generated code whereas space efficiency refers to the amount of space the generated code and run-time data take up. Space efficiency is important in applications where memory resources are restricted such as embedded systems. Time efficiency is desirable in most other applications. Note that the most space efficient code is not necessarily the most time efficient and frequently a compromise is made between the two.

Finally, efficient code generation refers to the speed at which target code is generated by the compiler. One factor affecting the speed of target code generation is the desired level of efficiency required of the target code. Increasing the quality of the target code typically slows down code generation as a result of the increased analysis of either the source text or some intermediate representation of the program. Another factor, which is the subject

of this thesis, that reduces the speed of target code generation is the redundant processing of source text and other data. Specifically, we are concerned with the case where the result is discarded either because it is irrelevant or it duplicates a previous result. The next section discusses redundancy in the context of separately compiled languages.

1.2 Separate compilation

The alternative to separate compilation could be described as monolithic compilation where the smallest unit a compiler can process is the entire source text of a complete program (see Figure 2). The figure directly illustrates the notion that several programs may

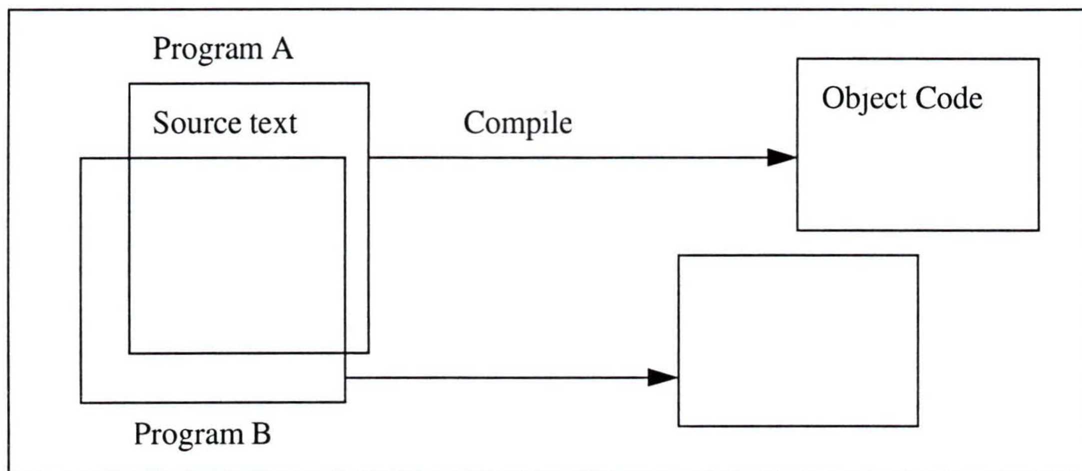


Figure 2. Monolithic compilation

contain common textual fragments, such as procedures or functions, which are compiled each time they encountered in a program, usually generating the same target code in the process. Another obvious inefficiency occurs when the text of the program changes in a small area and the text of the entire program must be recompiled as result. The advantage of a monolithic system, at least for strongly typed imperative languages, is that every object in the program can be declared exactly once and object references can be type checked easily.

If a language supports separate compilation, incomplete program fragments can be compiled and assembled later during a second phase, linking, which resolves references to objects that are not defined in the varius compilation units (see Figure 3) With separate

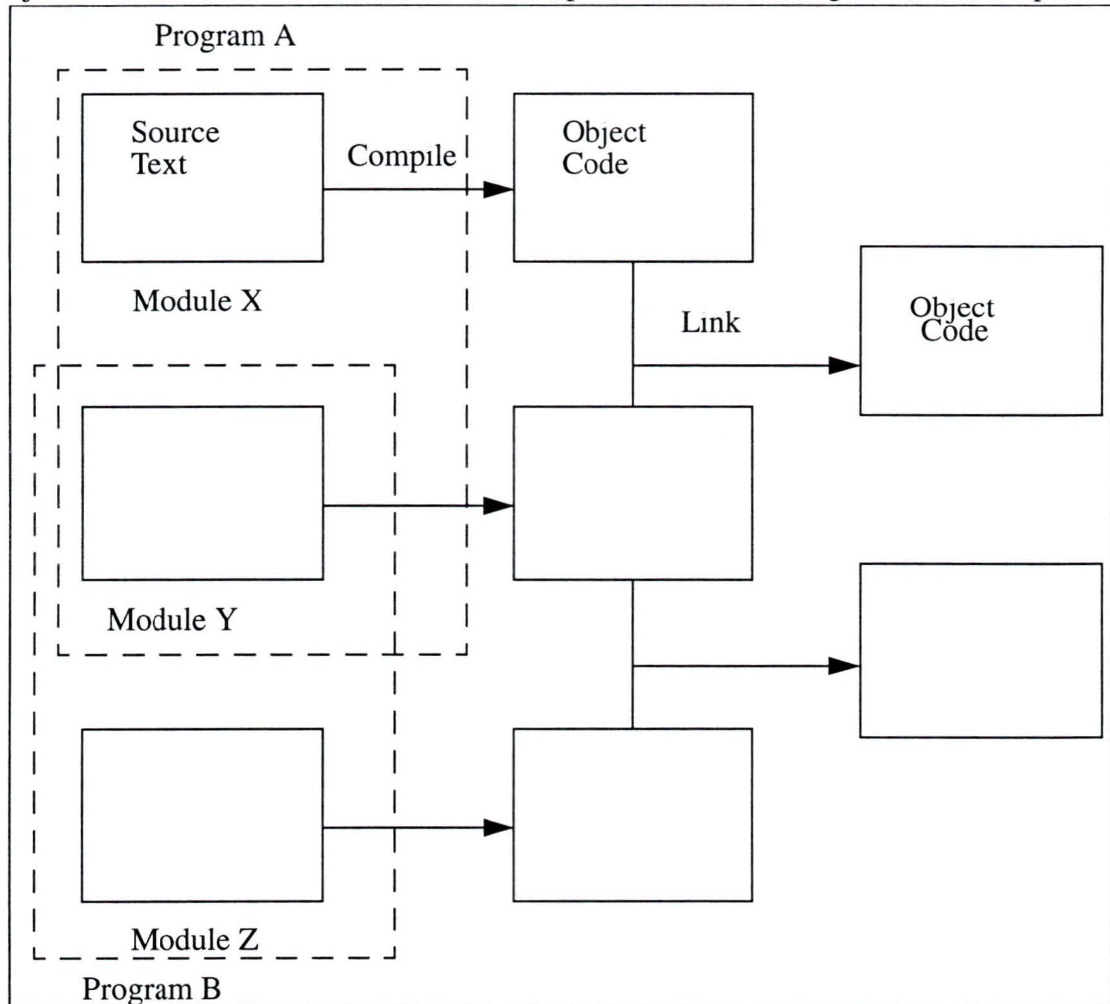


Figure 3. Separate compilation

compilation, common procedures and functions can be collected into one or more modules and compiled exactly once rather than once for each program they are used in The drawback is that the compiler now requires a mechanism for verifying the use of objects declared in one module and referenced in one or more other modules

C is properly called "independently compiled" rather than "separately compiled" since the language definition does not describe a formal mechanism for type checking between

independently compiled units. C's `#include` directive, however, can be used to *simulate* intermodule type checking if used properly. Moylan [15] gives a summary of the correct use of the `#include` directive for intermodule type checking.

The `#include` directive simply instructs the compiler to interrupt processing the text of the current file and start processing the file named in the directive. When the named file is finished being processed, processing continues in the original file. Inclusion directives can be nested. If we restrict the contents of included files to containing only declarations of objects appearing in the interface of modules then we can simulate intermodule type checking by including these files in any compilation unit that references an object in the interface of a particular module.

1.3 The problem

The central problem with independent compilation in C is that the text constituting the interfaces of modules is processed repeatedly in the case where a compilation unit is compiled repeatedly, as during the compile-debug-edit cycle, or when several compilation units are compiled which reference the same interface. In the days when C was in its infancy, when it was used largely for implementing the collection of small programs that comprised the Unix operating system, this problem was manageable since the number of modules in the system was small and the interfaces were simple. Today, as the complexity of software being implemented in C increases, the number and complexity of interfaces is increasing and, furthermore, the number of declarations actually used by a compilation unit is getting smaller relative to the total number of declarations appearing in included header files. Kamel [12] reports that, for one large system, 60 percent of compile time was spent processing interfaces. One might consider addressing the second problem by restricting each header to contain exactly one declaration and thus each compilation unit could include exactly the right number of headers for containing all the required declarations. This has the obvious disadvantage that it would be extremely difficult to keep the set of files required by a compilation unit up to date by hand, and, furthermore, the overhead

of opening and closing several small files for a single compilation unit would likely be a significant factor in slowing down compilation for most operating systems. At the other extreme we could keep all declarations in a single file which makes the second problem as bad as possible since the ratio of declarations processed to declarations used is as high as possible; on the positive side each compilation unit only ever has to include one header. With one monolithic header file we also run into the problem that C only has a single global name space for functions and variables and so a scheme is required to name every exported object of every module uniquely. In practice, declarations are grouped into header files based on some meaningful criteria for the programmer.

In light of the above discussion we now reconsider the sample program of Figure 1 more carefully. The five header files that are included transitively reference 15 other header files. On our system those seemingly innocuous five lines actually correspond to 285,000 bytes representing hundreds of declarations of which only 20 are even referenced by the function being compiled. The implication of this analysis is that the average number of instructions executed per byte read or written is only about 140 (based on 4 seconds to compile on a 10 MIPS machine).

Looking even more closely at the compilation process we discover that the compiler is actually implemented as two separate programs: the preprocessor and the compiler proper. The preprocessor performs part of the translation and outputs preprocessed C text where the `#include` directives have been expanded, macros have been expanded, and comments have been stripped out. For our sample program the output of the preprocessing phase alone is about 56,000 bytes. So the preprocessor reads 285,000 and writes 56,000 bytes and the compiler proper reads 56,000 and writes 2,700 bytes for a total of 399,700 bytes. The four second compile time now seems even more reasonable implying an average of 100 instructions executed per byte read or written.

Considering the actual amount of work being done by the compiler, it no longer seems that inefficient. However, we doubt that it has to do that much work all the time; the compiler essentially behaves like the imaginary translator in the opening paragraph of this chapter.

What we would really like is a compiler which behaves more like a typical human translator: someone who only looks up unknown words as they are encountered in the document being translated and remembers definitions between translations. This thesis examines the implementation of similar behavior in a C compiler.

1.4 Thesis overview

In Chapter 2 we summarize previous work that has been done to overcome the problem outlined in this chapter, but first look at the related problem of eliminating the redundant generation of object code. An overview of what we would like to achieve in general along with what is practical appears in Chapter 3. Chapter 4 describes some of the details of applying our approach to an existing compiler. Our modified compiler is measured against the original implementation in Chapter 5. Finally, Chapter 6 summarizes the effectiveness of our approach and outlines possible future work for this implementation as well as speculating on the general applicability of the technique to other programming languages.

2 Background

Extending the analogy of Chapter 1, suppose the book is so successful that it goes to a second printing. Naturally the author will take the opportunity to make some corrections, perhaps the corrections are confined to a single chapter or section. The translator reports that he cannot really guarantee the quality of the translation unless he is allowed to translate the entire book rather than just the changed sections. Obviously the amount of grief this causes is a function of the length of the book.

The approach of the translator may sound more outrageous than the original scenario, yet the cautious programmer often relies on exactly this, recompiling the entire program, in order to ensure that any changes to the program text have their effects propagated through the entire program. The reason is that dependencies exist among the compilation units comprising a complete program. It may not even be just the cautious programmer who recompiles the entire program, the sloppy, impatient programmer may recompile the entire program in the hope of removing a bug caused by a hidden dependency which he is unaware of. Recompilation of the entire program after a small change may not be an issue if the program is small but becomes less attractive as the size of the system increases.

This problem is actually quite similar to the problem we are primarily considering and described in Chapter 1. In this case we want to avoid regenerating the same object code, of which we already have a copy in a file, which will be used later on as input to the linker. For the problem described in Chapter 1, we would like to avoid regenerating the same

symbolic information from header files, of which we *should* retain a copy in memory, which can be thought of as input to the remainder of the compilation. This chapter summarizes the work that has been done in both areas.

2.1 Eliminating redundant compilation of modules

Adams et al. [1] provide an excellent summary of current techniques used in eliminating redundant compilation of modules, which they term selective recompilation and eliminating redundant compilation within modules, which they term efficient environment processing. Selective recompilation is concerned with establishing the set of units that are potentially affected by a particular change to the text of one unit. As a running example to be used throughout this section, we consider the program fragment of Figure 4. The explicit dependencies created by the `#include` directives are illustrated in Figure 5.

```
/* types.h */
typedef struct {
      double   x;
      double   y;
} RectPt;
typedef struct {
      double   rho;
      double   r;
} PolarPt;

/* modvar.h */
#include “types.h”
extern RectPt p1;

/* f1.c */
#include “modvar.h”
RectPt p1;
void f1( void ) {
      p1.x = 1.23;
      p1.y = 4.56;
}

/* f2.c */
#include “modvar.h”
void f2( void ) {
      p1.x += 2.0;
      p1.y += 3.0;
}

/* f3.c */
#include “types.h”
void f3( void ) {
      RectPt p;
      p.x = 2.0;
      p.y = 3.0;
}

/* f4.c */
#include <math.h>
#include “types.h”
void f4( void ) {
      PolarPt p;
      p.rho = M_PI/3.0;
      p.r = 2.0;
}
```

Figure 4. Program fragment with dependencies

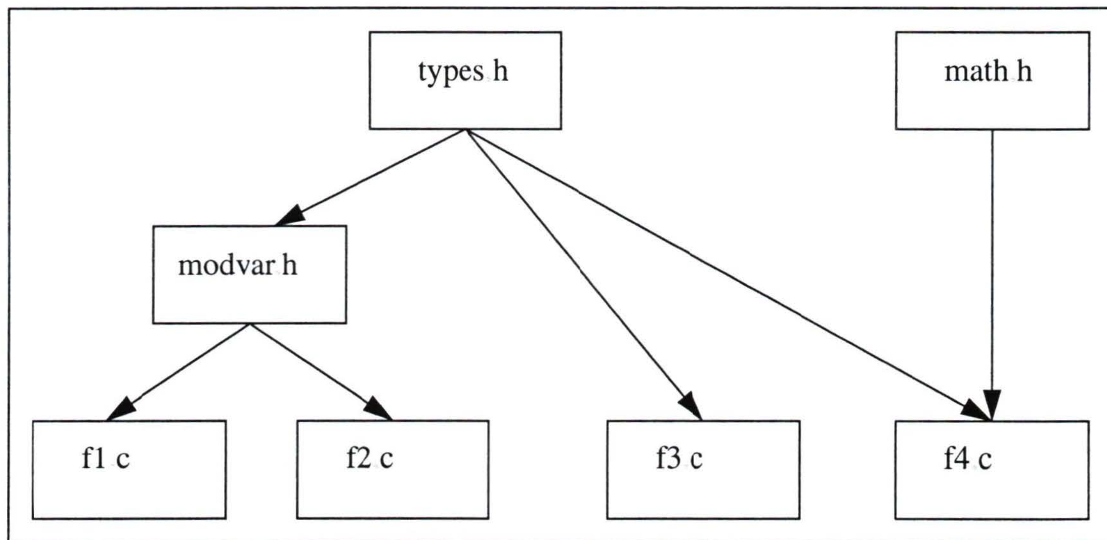


Figure 5. Dependency graph for program of Figure 4

The eight methods of selective recompilation identified by Adams et al are cascading recompilation, surface change, cut-off recompilation, semi-smart recompilation, smart recompilation, attribute recompilation, smarter recompilation, and smartest recompilation There is a ninth, not very selective, method termed the *big bang* method where the entire system is recompiled when a single unit changes

2.1.1 Cascading recompilation

The rule for cascading recompilation is simple whenever a unit changes, all of its directly and indirectly dependent units are recompiled In implementations where specification units are actually compiled (e g Ada and Modula-2 compilers), units are compiled in the order of increasing distance in the dependency graph from the changed unit In implementations where specification units are not compiled (typical C and C++ compilers and the GNU Ada compiler) only the leaves on a path from the changed unit are recompiled For our example program fragment, if `types h` changed then all the " `c`" files would be recompiled, if `math h` changed only `f4 c` would be recompiled, if `modvar h` changed only `f1 c` and `f2 c` would be recompiled `Make` [4] is a tool designed to identify which units in a hierarchy have been changed and schedule recompilations accordingly Historically, the hierarchical descriptions of programs were maintained by hand but now, at least

for some languages, there are tools for generating the descriptions automatically from the program text

Under strict cascading recompilation, the nature of the change to a unit is considered irrelevant It is clear that not every change affects every dependent unit and so several refinements to the method have been made

The surface change method analyzes the changed unit to establish if the change is strictly cosmetic, i e change is confined to comments and white space, if so, there is no need to recompile dependent units If, for example we add a comment to `types h` then no units need to be recompiled This analysis can be implemented easily by retaining a tokenized version of the compilation unit and retokenizing the unit when the text changes and comparing the sequence of tokens

The cut-off recompilation method establishes the set of units to be recompiled in the same manner as cascading recompilation The output of the compilation of each unit is compared with the previous output In the event that there is no change, no further dependants of this unit need to be recompiled This method is not particularly applicable to C where specification units (" `h`" files) are not explicitly compiled but could be applied to languages like Ada or Modula-2 Consider the equivalent of the example program in a suitable language and the addition of a field to the PolarPt structure Cascading recompilation would select all units for recompilation The output of compiling modvar h would be no different from the previous compilation and so `f1 c` and `f2 c` would be "cut-off" from further recompilation

2.1.2 Smart recompilation

Smart recompilation methods analyze the nature of changes to compilation units in greater detail Semi-smart recompilation classifies the change to the unit as either adding declarations to an exported interface, modifying procedure declarations or variables, or other changes For some languages (not C, however) where the exported objects of an interface of a module are qualified by the module name then the addition of declarations to an inter-

face will have no effect on any directly or indirectly dependent units. Also if the language allows variables and procedures to only propagate one level (also not C) then only directly dependent units can be affected by a changes to procedure declarations or variables. All other changes generate cascading recompilation. As an example consider the sample program in a language supporting qualified object references and the addition of the following declaration to `types.h`:

```
typedef struct {
      PolarPt        centre;
      double         radius;
} Circle;
```

In this case no units would be recompiled because no unchanged units can possibly depend on an added declaration. For C of course we would have to recompile all files transitively dependent on `types.h` since the added identifier Circle may clash with another declaration of Circle since all exported objects of modules in C end up in the same global name space.

Smart recompilation [20] is more precise in its analysis. For each declaration in each compilation unit the set of compilation units referencing that object is computed. This information is used in conjunction with the sets of added, modified, and deleted declarations when a compilation unit is changed in order to determine the set of compilation units that must be recompiled. For the sample program fragment we have:

Compilation unit	Declaration	Transitively referenced in
types.h	RectPt	{ modvar.h, f1.c, f2.c, f3.c }
types.h	PolarPt	{ f4.c }
modvar.h	p1	{ f1.c, f2.c }
f1.c	f1	{}
f2.c	f2	{}
f3.c	f3	{}
f4.c	f4	{}

Consider a change to a compilation unit that is syntactically valid. Dealing with the deleted declarations is easy, either the declaration is not referenced by any other unit, in which case the deletion is harmless and no other unit needs to be recompiled, or the declaration is referenced elsewhere, in which case an error has been introduced since there will be a reference to an identifier somewhere without a declaration. Dealing with added and modified declarations is slightly more complicated. Firstly, both added and modified declarations may contain references to identifiers not explicitly declared in the changed compilation unit in which case a recompilation is necessary to establish that the use of the free identifiers is valid. Added declarations must be checked against all other units to establish if they redeclare identifiers declared in other units. Finally, the set of units that reference a modified declaration must be recompiled in order to ensure that uses of the declared identifier are still valid.

As an example, consider the addition of the declaration of Circle above to types.h. This is not an error since no other unit declares Circle and also nothing needs to be recompiled. As a further example consider the addition of a field, `z`, to `RectPt`, in this case `mod-var.h`, `f1.c`, `f2.c`, and `f3.c` would be recompiled because they all reference `RectPt`.

There is an efficient graph based implementation of smart recompilation [11] in which vertices represent modules and arcs represent dependencies between modules. The arcs are labelled with the objects that are exported directly or indirectly to the module at the other end of the arc. The algorithm performs a traversal of the graph from the changed module in order to establish the set of modules that need to be recompiled.

Attribute recompilation refines the set of referencing units for a declaration to include more information regarding the nature of the use of the identifier. For example, if there is an identifier that names a record type and the name is only referenced to get the size of the type then changing the fields of the record should not force a recompilation (provided the size of the record stays the same). This is not illustrated in the example program fragment

but suppose there was an additional compilation unit:

```
/* f5.c */
#include "types.h"
void f5( void ) {
     int s = sizeof( PolarPt );
     ...
}
```

Now, if we renamed a field in the `PolarPt` record type that would constitute a change to the declaration of `PolarPt` so under smart recompilation `f5.c` would have to be recompiled. Attribute compilation would have the information that the reference to `PolarPt` in `f5.c` only uses the size of the type and so the renaming of a field would not cause the recompilation of `f5.c`.

Smarter recompilation [17] recognizes that symbols are frequently used independently in different parts of large system. For example, consider again adding a field, `z`, to the `RectPt` type in the sample fragment and adding a statement to `f1.c` to initialize the `z` field of `p1`. Smart recompilation would select `modvar.h`, `f1.c`, `f2.c`, and `f3.c` for recompilation. The programmer knows that only a new field was added to the type `RectPt` and that this new field is only used in `f1.c` so she selects the subset {`modvar.h`, `f1.c`} for recompilation. The smarter recompilation algorithm would identify `f2.c` as sharing an object of the modified type with `f1.c` and so `f2.c` would be added to the set of units to be recompiled as well. The unit `f3.c` need not be recompiled if it contains an independent use of the modified type. This method has the benefit that a change can be tested on a smaller part of a larger system without recompiling the entire system while still creating a functional system. If the test is successful then the entire system can be recompiled at a later time. The general process of smarter recompilation is:

1. The changed units are analyzed to form the set of units that would be selected for recompilation by the smart recompilation method and presented to the programmer.
2. The programmer selects a subset of these to be recompiled.
3. The compilation system infers any additional units that must be recompiled that share objects of modified types with the units selected by the programmer for

recompilation Shared objects typically arise as module wide variables of modified types and procedure calls with arguments of modified types

Finally, smartest recompilation, normally only applicable to languages with no explicit specification units such as ML, only recompiles changed implementations, infers the most general types for undeclared identifiers and defers type resolution between modules until linking

2.2 Eliminating redundant compilation within modules

Adams et al [1] identify two methods for eliminating redundant compilation within modules selective embedding and environment pruning Selective embedding ignores module interfaces until the set of references to undeclared identifiers appearing in a compilation unit has been determined At this stage the required declarations are extracted and compiled This process might uncover more undeclared identifiers and so the process is iterated until every referenced identifier has had its declaration compiled

Environment pruning is a more coarse approach to selective embedding Module interfaces are again ignored until the set of referenced identifiers without declarations has been determined For each undeclared identifier the entire interface containing its declaration is processed This technique relies on references to external symbols being qualified so the interface can be identified easily, otherwise all interfaces specified by the compilation unit must be processed

2.2.1 Precompiled headers

A precompiled header is an intermediate form of a header file intended to be processed more quickly by the compiler than the source form of the header The Lattice C compiler for the Amiga [13] has a utility which removes unnecessary white space and comments from a header and replaces keywords with single character representations (distinguished from the regular text by having the high order bit set) This processed text still has to be parsed and analyzed like an ordinary header However the volume of input is much smaller in most instances This compiler also has a mode of operation whereby the symbol

table of a compilation can be dumped into a file and read during subsequent compilations. Several of these precompiled symbol files can be read in during a compilation before the source file is processed. However it is up to the programmer to ensure that each symbol has a unique definition, the compiler does not seem to have a mechanism for checking consistency among several symbol files.

The THINK C compiler for the Macintosh [19] can also generate symbol files containing declarations in a format suitable for quickly loading into the compiler's symbol table. Consistency issues are handled by restricting source files to only include a single precompiled header and that header must be the first file included. The Borland C++ compiler [2] has a similar mechanism and deals with consistency by restricting the use of precompiled headers to source files where the same set of headers is included in the same order and the same set of macro definitions must be in effect.

Litman [14] describes a more flexible approach whereby the precompiled header contains an index of locations of declarations in the source of the original header keyed by identifiers as illustrated in Figure 6. This mechanism is used instead of a dump of the symbol table of the compiler. The precompiled header also contains the set of macro definitions of identifiers appearing in the header when it was precompiled which can be compared for consistency against the definitions in effect when the header is included. Two situations

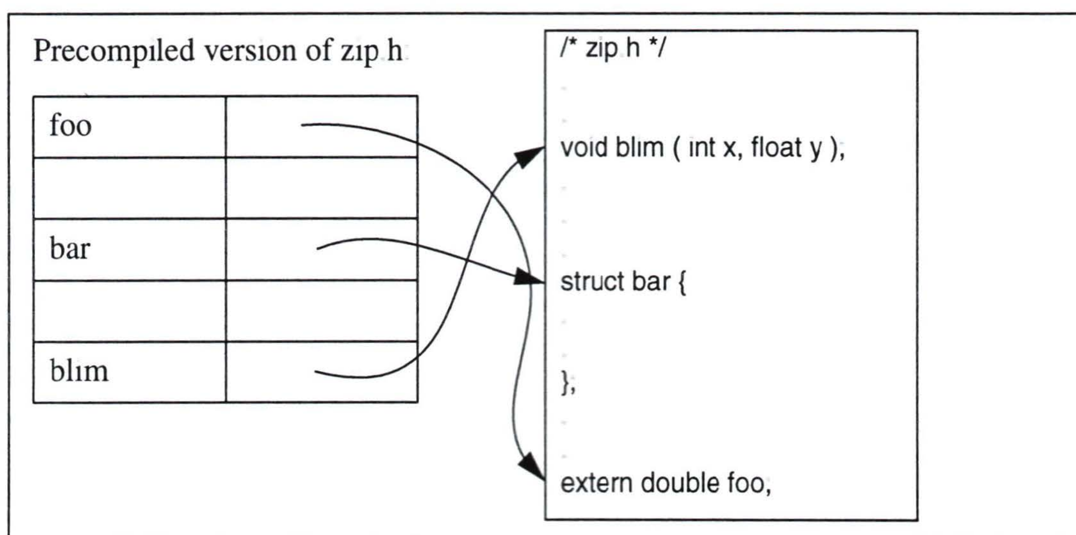


Figure 6. Litman's precompiled headers

are identified which make the stored context and the current context incompatible The first case is that a macro was defined in the stored context that is not defined in the current context This is problematic only if the macro is referenced somewhere in the text of the header and consequently the stored header contains a referenced flag for each defined identifier to enable this test The second case is that a macro is defined in the current context but not in the stored header Again this is a problem only if the macro in question is actually referenced in the text of the header In this case the set of identifiers appearing in the header is searched for the macro and in the event it is found the non-precompiled version of the header is used instead The check is conservative but in practice does not appear to eliminate the use of the precompiled header too often The second contextual problem, any remaining references to undeclared identifiers after preprocessing, does not cause a problem since the extracted declarations are processed in their source form so any references will be looked up in the usual way Although not discussed in the paper, we assume Litman's implementation will compile programs that do not conform to the language standard For, example if a file includes two headers with conflicting definitions for a symbol then no error is reported provided the symbol is never referenced in the file because the text of the declarations will never be processed We discuss this kind of "harmless" non-conformance later

A sophisticated preprocessor records when a precompiled header is available for each header included in the source file When a reference to an undeclared identifier is encountered later in the source file the indices of the precompiled headers are searched The text of the appropriate declaration is extracted from the source file and copied to the output stream The output of the preprocessor is compiled as usual The author reports that this technique has reduced overall compile time for projects by 25 to 65 percent Note that since all declarations are processed in their source form by this method it requires a low use to visibility ratio for the method to actually save time If every declaration is used by every compilation unit then this method could not possibly save time over conventional compilation methods since the technique has the additional overhead of looking up declarations in the precompiled header as well processing the text of the declaration itself The

case of every declaration being used by every compilation unit should not occur often in practice.

2.2.2 Compilation servers

Onodera [16] presents an interesting approach which does not strictly correspond to either selective embedding or environment pruning. The central idea is to turn the compiler into a server process which saves the internal representation (IR) of a header file in memory between compilations and reuses this representation when the header file is encountered in subsequent compilations. The approach is applied to COB, a C based object oriented language.

Two situations are identified which are expected to benefit from this scheme: *massive* and *repetitive* compilation. Massive compilation occurs when several source files are selected for recompilation, perhaps as a result of a change to a high level interface. Repetitive compilation occurs during the edit-compile-debug cycle when the same source file is repeatedly compiled. Perhaps a small part of the file is changed each time. Imported interfaces typically remain unchanged.

Header files in COB are partitioned into two groups: *interface files* and *declaration files*. Declaration files are used in the usual way and can contain preprocessing directives. Interface files consist solely of class declarations, cannot contain preprocessing directives, and are included on demand rather than explicitly by the `#include` directive. When a class is referenced for the first time in a compilation unit, the compiler suspends processing of the main text, finds and processes the corresponding interface file and then resumes processing of the main file. The COB compiler is restricted to only retaining the internal representation of interface files. This simplifies a number of issues. Firstly, the compiler does not have to check for compatible preprocessing contexts since interface files are not subject to preprocessing. Secondly, any exposed identifier remaining in an interface file will have a definition in another interface file and so its definition is the same across compilations because of the rules for processing interface files.

In particular the compilation server has a symbol table which is never completely "emptied" between compilations. Any symbol definitions which were entered into the symbol table as a result of processing the text of interface files are retained between compilations. The symbol definitions can be reused indefinitely often in subsequent compilations without re-reading the interface file as illustrated in Figure 7. The only time symbol definitions from interface files would need to be removed from the symbol table is when the text of the interface file is updated.

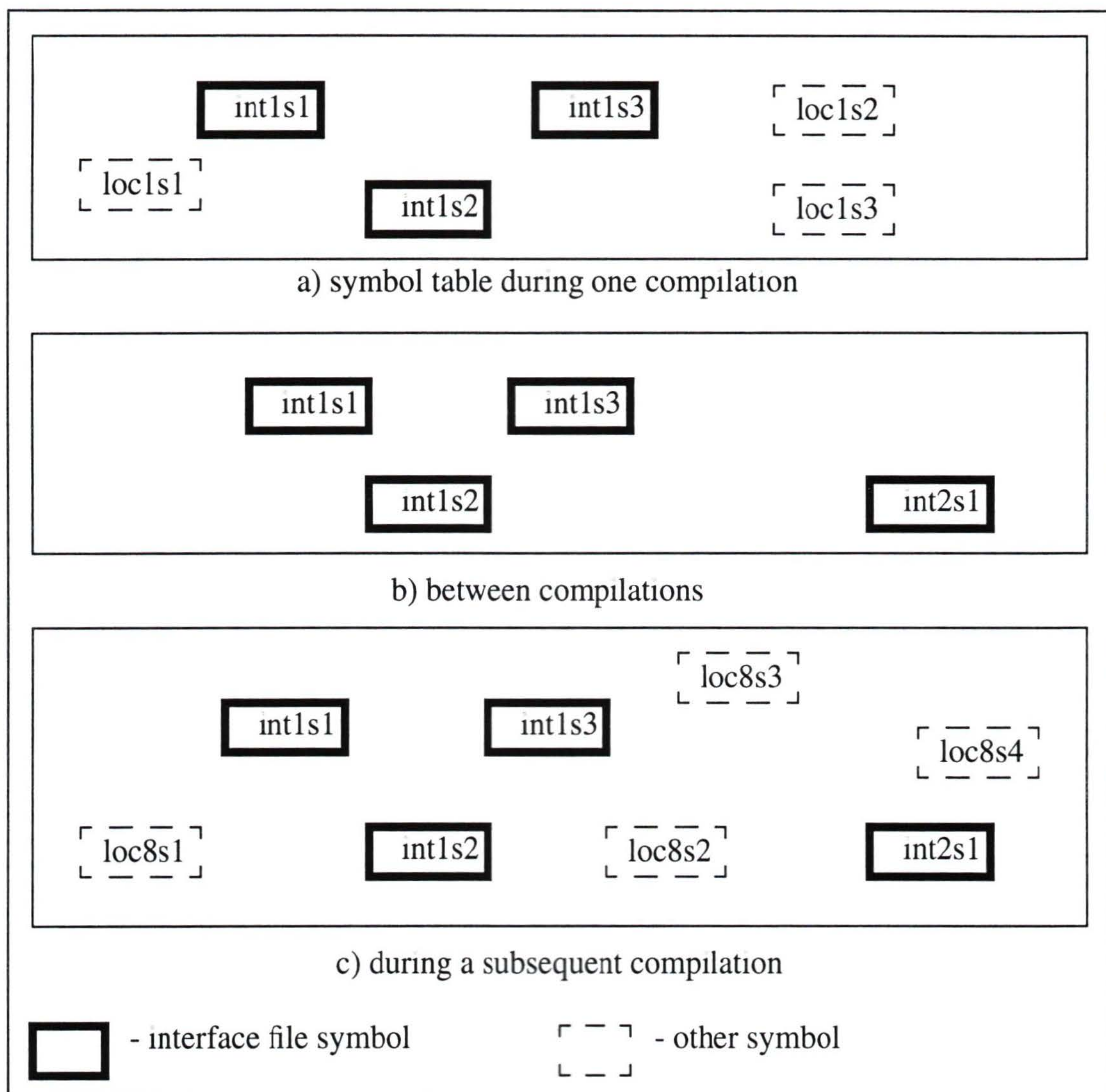


Figure 7. COB symbol table

The COB compiler is written in COB, a language whose runtime environment includes automatic garbage collection. At the end of processing a compilation unit, all structures not corresponding to the internal representation file are marked as garbage. The compiler also detects when an internal representation has become out of date with respect to the source text and purges it along with any dependent interfaces. Subsequent compilations reuse the internal representations of class interfaces provided they exist and are up to date.

The author reports real time savings of 40% when compiling the entire compiler (massive case) and 50 to 70% in the repetitive case. Note that, in contrast to Litman's approach, this technique may not break down when the use to visibility ratio gets too high provided the overhead involved in retaining symbol table information between compilations is not more expensive than processing the text of headers in the conventional way.

Fyfe et al. [6] describe how conventional compilers have been converted into compilation servers. Their implementation does not keep the internal representation of header files in the address space of the server processes. Instead the representation of a header file gets stored in object files. Sharable representations of several header files may be stored in a separate object file for reference by all compilations that contribute to it. During subsequent compilations, the symbol table information can be read from the appropriate segments of the appropriate object files instead of processing the text of the header file. Their implementation also only reads symbol information for symbols that are actually used by a compilation unit rather than for all the symbols in an file. They do not address the problem of dealing with different contexts which may change the meaning of the contents of the header file.

2.3 Summary

This chapter has surveyed a number of techniques for establishing the minimum number of compilation units that are affected by a change to the program text. Approaches to minimizing the amount of work required for processing the interface needed by a compilation unit have also been covered. We conclude by remarking that an efficient, complete pro-

gramming environment is likely to employ both techniques in sequence, minimizing the number of compilation units to be recompiled after a change and then compiling those units selected as efficiently as possible. It is the second step which forms the basis for the remainder of this thesis.

3 Approach

This chapter describes, in general terms, the approach intended to be used in implementing a compilation server for C similar to the COB compilation server outlined in the previous chapter. Our approach differs primarily in terms of which header files are considered for retention in the compilation server and the mechanism by which symbols are moved in bulk in and out of the symbol table.

The COB language distinguishes between conventional header files which are explicitly included in the usual way by the use of the `#include` directive, and interface header files which are included implicitly when a reference to a class appears in the program text. The COB compiler only retains the internal representation of interface files. The implication is that symbol definitions which come from interface files can be left in the symbol table indefinitely until the text of the interface file changes at which all symbols declared by the file would have to be removed. In contrast our implementation attempts to retain the internal representation of conventional header files. Since conventional header files are included in program text explicitly, we need a different mechanism for moving large number of symbol definitions in and out of the symbol table efficiently.

3.1 Equivalence of C program fragments

At the most abstract level, a C compiler computes a function mapping the text of a compilation unit to the corresponding target text, usually either assembly or object code. The source text has some structure and in practice the text of a compilation unit is processed one source language construct at a time (see Figure 8). For C compilers, the structure of

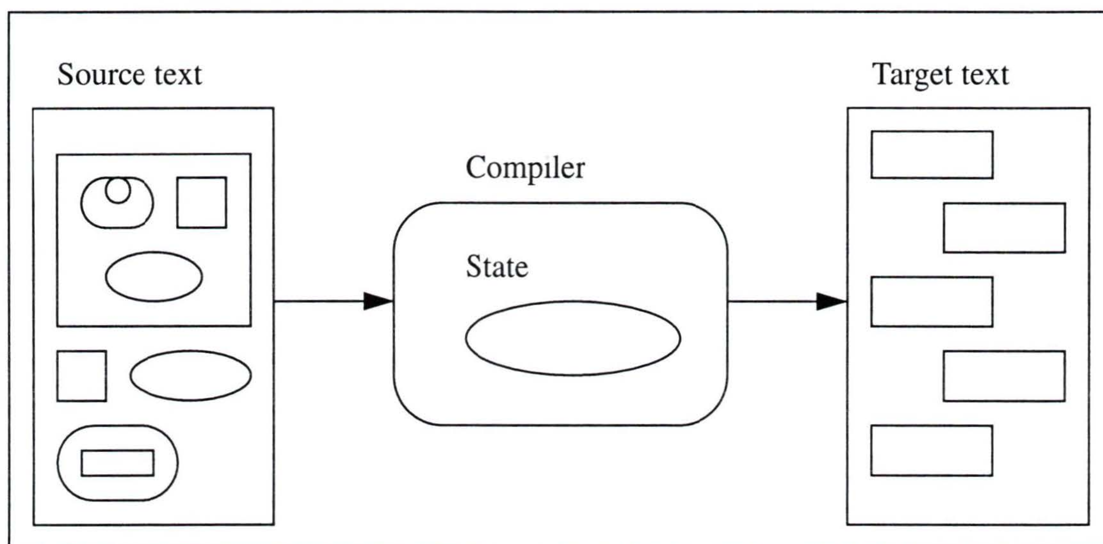


Figure 8. Structure of conventional compilation

the source text is much richer than the structure of the target text. The effect of processing a construct is a function of the text and the state of the current compilation. The effect may involve updating the state of the current compilation, generating a piece of the target text, or both. The goal is to implement a compiler which can identify fragments of text that will generate the same effect as fragments previously processed and duplicate the effect. On average we would like the cost of identifying the fragment and duplicating the effect to be less than processing the fragment in its source form. We define a pair comprised of compilation state and text fragment as equivalent to another pair if they both generate the same output and change in compilation state.

3.1.1 Compilation state

The compilation state can be thought of as a set where each element associates a meaning with an identifier. An identifier in a C program can be defined to represent a sequence of tokens which should replace the identifier whenever it appears in the program text, or it labels some object of a program such as a variable, function, or type. As identified in the introduction, the compiler is typically split into two processes, the preprocessor and the compiler proper and so compilation state is split accordingly.

3.1.2 Preprocessing state

The preprocessor takes C text as input and generates preprocessed C text as output. The compiler proper takes preprocessed C text as input and generates assembly or object code as output. The essence of preprocessing state is a set of macro definitions and a location in the input. Because of nested `#include` directives, the location in the input may be represented by a stack of positions in several files. A macro is a triple consisting of an identifier, the number of formal parameters the macro takes, and a replacement sequence of tokens, possibly empty. The set contains at most one definition per identifier. Definitions become part of the set as the result of processing `#define` directives appearing in the source text, for example, the lines:

```
#define ARRSIZE 100
#define Slope(p1,p2) ((p2.y-p1.y)/(p2.x-p1.x))
```

define the two macros:

<ARRSIZE, 0, <100>>, and

<Slope, 2, <(,(,@2,.,y,-,@1,.,y,),/,(,@2,.,x,-,@1,.,x,),)>>

respectively. The names of the formal parameters of a macro are irrelevant and so we have replaced them with pseudo-tokens of the form @*n*, where @1 corresponds to the first formal parameter, @2 corresponds to the second formal parameter, and so on. The effect of a macro definition is that subsequent occurrences of the defined identifier, and any arguments immediately following, are replaced with the sequence of tokens given in the definition with the actual parameters replacing the formal parameters, for example:

```
Point Polyline[ARRSIZE];
slp = Slope( Polyline[7], cursor->loc );
```

is replaced with:

```
Point Polyline[100];
slp = ((cursor->loc.y-Polyline[7].y)/(cursor->loc.x-
Polyline[7].x));
```

Definitions are removed from the set as the result of processing `#undef` directives appearing in the source text. Macro definitions also affect the output of conditional compilation directives, for example:

```
#if ARRSIZE > 50
     short vec[ARRSIZE];
#else
     long vec[ARRSIZE];
#endif
```

is replaced with:

```
     long vec[30];
```

if, for example, ARRSIZE is defined to be 30 and with:

```
     short vec[75];
```

if ARRSIZE is defined to be 75.

3.1.3 The state of the compilation proper

The essence of the state of a compilation is a set of symbol definitions and the current location in the input. The elements of the symbol definition set are more complex than a macro definition. A symbol either: names a variable, function, formal parameter, enumeration literal, type, or member of a structure or union, labels a statement, or is used as a tag for a structure, union, or enumeration type. Unlike macro identifiers which remain defined until the end of the translation unit or until a corresponding `#undef` directive is encountered, each symbol is associated with a particular scope, either the entire compilation unit or a particular block. Since blocks can be nested, a local symbol hides a symbol with the same name in an enclosing scope. Furthermore, each scope is partitioned into separate

name spaces so symbols in one scope can use the same name provided they exist in different name spaces. A separate name space exists in each scope for structure, union, and enumeration tags, the members of a particular structure or union, labels, and all other identifiers.

3.2 Identifying equivalent fragments

At the outset of this chapter we said that a compiler can be viewed as computing a function which maps text in a source language to text in a target language, $Comp : SText \rightarrow TText$ where the domain, *SText*, denotes the set of sequences of preprocessing tokens. A preprocessing token is either a header name, identifier, preprocessing number, character constant, string literal, operator, or punctuator as specified in the ANSI C language definition [23]. Additionally, we allow newline characters in the token sequences in order to distinguish the correct or incorrect usage of preprocessing directives. The domain, *TText*, denotes the set of target code sequences which is implementation defined; typically elements of this set are sequences of assembly language instructions; this domain also includes sequences containing a special error token, *err_tt*, used for indicating invalid source sequences. The function *Comp* maps source sequences to target sequences; if an element of *SText* constitutes a valid translation unit according to the language definition then the sequence is mapped to the corresponding target sequence; if the element does not constitute a valid compilation unit then the sequence is mapped to a target sequence containing one or more occurrences of *err_tt* by definition.

Since the preprocessing and compilation proper phases are usually separated we could consider breaking *Comp* into two functions, $Prep : SText \rightarrow PText$, and $Compp : PText \rightarrow TText$ where the domain, *PText*, denotes the set of sequences of compilation tokens. A compilation token is either a keyword, identifier, constant, string literal, operator, or punctuator as specified in the language definition; this domain also includes sequences with error tokens, *err_pt*, used for signaling invalid preprocessing sequences. The function *Prep* maps sequences of preprocessing tokens to sequences of compilation

tokens, in the event that the preprocessing sequence is not valid according to the language definition then the entire sequence is mapped to a sequence containing one or more occurrences of *err_pt* by definition. The function *Compp* maps sequences of compilation tokens to target sequences; if the source sequence is invalid then it is mapped to a sequence containing one or more occurrences of *err_tt* by definition, also *Compp* maps a sequence containing *err_pt* to a sequence containing *err_tt* by definition.

These functions describe how the compiler maps the entire text of compilation unit. In practice the text is processed a fragment at a time and the text in conjunction with that part of the compilation state which represents the definitions of identifiers determines the output. So the three functions above could actually be computed with the related fragment functions:

$$CompF : SText \times PState \times CpState \rightarrow TText \times PState \times CpState$$

$$PrepF : SText \times PState \rightarrow PText \times PState$$

$$ComppF : PText \times CpState \rightarrow TText \times CpState$$

Here, *PState* denotes the domain of macro definition components of preprocessor states. As mentioned earlier, the set of macro definitions can be represented as a set of triples:

{ <Ident, Arity, Body > }

where Ident represents the name of the macro, Arity is the number of formal parameters required by the macro, and Body is a sequence of zero or more tokens with the pseudo-tokens, @*n*, representing the formal parameters. The elements of the domain are subject to the following constraint:

$$\begin{aligned}
&\forall ps\ (\ ps \in PState\ ) : \\
&\quad \forall md1, md2\ (\ md1, md2 \in ps\ ) : \\
&\qquad macid(\ md1\ ) = macid(\ md2\ ) \rightarrow md1 = md2
\end{aligned}$$

where *macid* is a function returning the identifier defined by a macro (i.e. the first element of the triple, $macid\ (\langle x, y, z\rangle) = x$). *PrepF* is thus a function mapping pairs of source text sequences and preprocessor state to pairs of preprocessed text sequences and (a possi-

bly different) preprocessor state. Again we have the convention that if a source text sequence is invalid input with respect to some preprocessor state (and the language definition) then the value of the function for that pair is a pair where *err_pt* occurs in the sequence that forms the first component of the pair.

CpState denotes the domain of symbol definition components of compiler proper states. The set of symbol definitions can be represented by a set of triples:

{ <Ident, Loc, AttrList> }

where Ident represents the name the of the symbol, Loc represents the scope and name space of the symbol, and AttrList is a list of attribute values. Loc is represented by a pair, <Scope,Space>, where Scope is either the global scope, the scope created for the declaration of parameters of each function, or the scope created for each compound statement within a function definition. Each scope is partitioned into name spaces and there is a separate space for each of: statement labels, the tags of structures, unions and enumerations, the members of each structure or union, and all other identifiers. The list of attribute values is implementation defined but will at least contain information such as: if a symbol is a tag, whether it is a structure, union, or enumeration tag; if a symbol is variable, whether it was declared `const` or `volatile`, its type, and so on. The elements of the *CpState* domain are subject to the following constraint:

$$\begin{aligned}&\forall cs\ (\ cs \in CpState\ ) \\ &\quad \forall sym1, sym2\ (\ sym1, sym2 \in cs\ ) \\ &\qquad symid(\ sym1\ ) = symid(\ sym2\ ) \wedge symloc(\ sym1\ ) = symloc(\ sym2\ ) \\ &\qquad\quad \rightarrow sym1 = sym2\end{aligned}$$

where *symid* and *symloc* return the identifier and location of the symbol respectively (i.e. the first and second elements of the triple, $symid\ (<x, y, z>)\ = x$ and $symloc\ (<x, y, z>)\ = y$). *ComppF* is a function mapping pairs of preprocessed text sequences and compiler state to target text sequences and (a possibly different) compiler state. We have the convention that if the preprocessed text sequence is invalid with respect to some compiler state (and the language definition) then the value of the function for that pair is a pair with *err_tt* as the value of the first element by definition. Also by definition,

ComppF maps any sequence containing *err_pt* to a sequence containing *err_tt*

CompF is just the composition of the functions *PrepF* and *ComppF*

As far as equivalency and a compilation server are concerned, we do not really care about the total output state of these functions but rather the change in state that they generate The functions above could be computed using the following functions which have change of state as output rather than the entire state

$$\Delta CompF : SText \times PState \times CpState \rightarrow TText \times \Delta PState \times \Delta CpState$$

$$\Delta PrepF : SText \times PState \rightarrow PText \times \Delta PState$$

$$\Delta ComppF : PText \times CpState \rightarrow TText \times \Delta CpState$$

Here, $\Delta PState$ denotes the domain of changes in preprocessor state A change in preprocessor state can be represented by a set operations that change the state of the preprocessor, the operations are either of the form

<**Define_Macro**, Ident, Arity, Body>

or

<**Undefine_Macro**, Ident>

which represent either the addition or the deletion respectively of a macro definition to or from the state of the preprocessor The elements of this domain are subject to the same kind of constraint as elements of *PState*

$$\begin{aligned}
&\forall dps\ (\ dps \in \Delta PState\ ) : \\
&\quad \forall op1, op2\ (\ op1, op2 \in dps\ ) : \\
&\quad\quad macid(\ op1\ ) = macid(\ op2\ ) \rightarrow op1 = op2
\end{aligned}$$

$\Delta PrepF$ is essentially equivalent to *PrepF* but contains only the operations required to update the state of the preprocessor rather than produce the entire resulting state of the preprocessor

$\Delta CpState$ denotes the domain of changes in compiler state A change in compiler state can be represented by a set of operations that change the state of the compiler, the opera-

tions are of the form

<**Add_Symbol**, Ident, Loc, AttrList>

Again, the elements of this domain are subject to the same kinds of constraint as elements of *CpState*

$$\forall dcs\ (\ dcs \in \Delta CpState\ )$$
$$\quad \forall op1, op2\ (\ op1, op2 \in dcs\ )$$
$$\quad\quad symid(\ op1\ ) = symid(\ op2\ ) \wedge symloc(\ op1\ ) = symloc(\ op2\ )$$
$$\quad\quad\quad \rightarrow op1 = op2$$

$\Delta CompppF$ is equivalent to *ComppF* but contains only the operations required to update the state of the compiler rather than produce the entire resulting state of the compiler

$\Delta CompF$ is the composition of $\Delta PrepF$ and $\Delta ComppF$

The basic concept is that every triple of text fragment, preprocessor state, and compiler state corresponds to some output text and some change in preprocessor state and compiler state This is intended to correspond closely to how a compiler operates in practice processing some textual fragment, outputting some target text, and updating state

The function, $\Delta CompF$ above, can be represented by a graph with two types of vertices, st_i representing elements of the set $SText$, and (tt_k, dps_l, dcs_j) representing elements of $TText \times \Delta PState \times \Delta CpState$ Edges only run from one type of vertex to vertices of the other type and are labelled with the set of states for which the text at one vertex generates the output/change in state at the other vertex A portion of the graph is shown in Figure 9 As a concrete example we consider the text fragments st_i and st_{i+1} to be

```
foo x;
```

and

```
bar x;
```

respectively If cs_m denotes the state

```
{ <foo, (global,other), <typedef name,int> > }
```

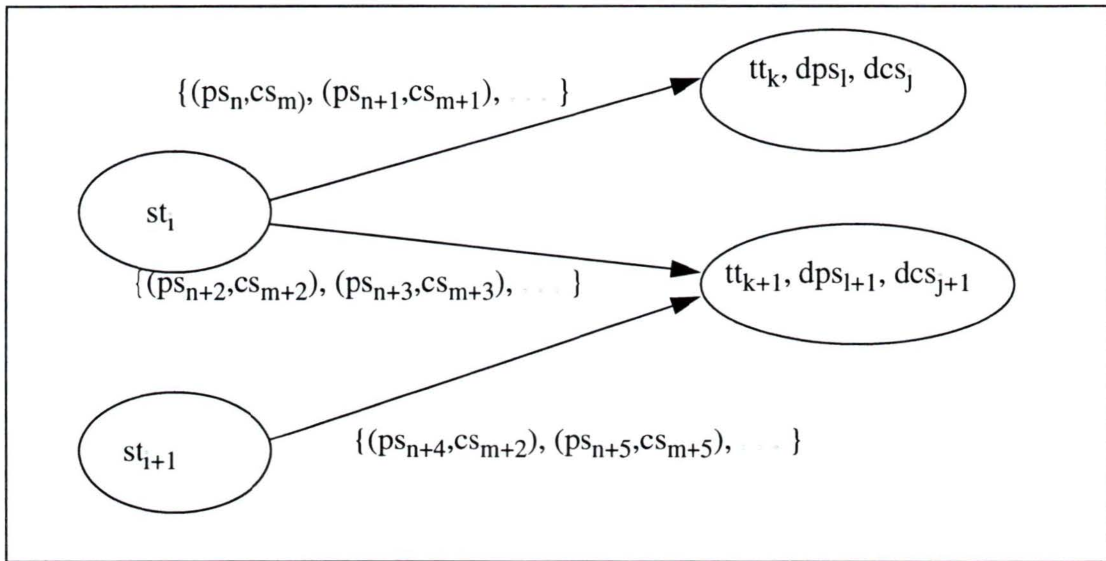


Figure 9. Relationship between source text, target text, and compilation state

and ps_{n+2} denotes the state:

```
{ <foo, 0, <blim>> }
```

and cs_{m+2} denotes the state:

```
{ <blim, (global,other), <typedef name, float> > }
```

and ps_{n+4} denotes the state:

```
{ <bar, 0, <blim> > }
```

then tt_k and tt_{k+1} both represent an empty output text sequence and the changes in state represented by dcs_j and dcs_{j+1} are:

```
<Add_Symbol, x, (global,other), <variable, int > >
```

and

```
<Add_Symbol, x, (global,other), <variable, float> >
```

respectively. Here the term **other** is used to indicate the symbol is in the normal name space (as opposed to the space for structure tags or statement labels, etc.), the term **variable** distinguishes the symbol from a function, **typedef name**, etc. The important thing to

note about the graph is that one source fragment can generate several effects depending on state and that one effect can be generated by several different source fragments depending on state. The functions $\Delta PrepF$ and $\Delta ComppF$ can also be represented by a graph with three types of vertices, st_i as above, (pt_o, dps_l) representing elements of $PText \times \Delta PState$, and (tt_k, dcs_j) representing elements of $TTextF \times \Delta CpState$. Edges only run from the first type of vertex to the second type and the second type to the third type. The edges are labelled with set of states which generates the effect at the destination vertex from the text at the source vertex. A portion of the graph is shown in Figure 10. In the context of the above example pt_o and pt_{o+1} could represent

```
foo x,
```

and

```
blim x,
```

respectively.

The general goal of a compilation server is to retain a subset of associations between input text and change in state, and identify when an upcoming text fragment along with the current state is equivalent to some previously processed pair. In the event that there is a

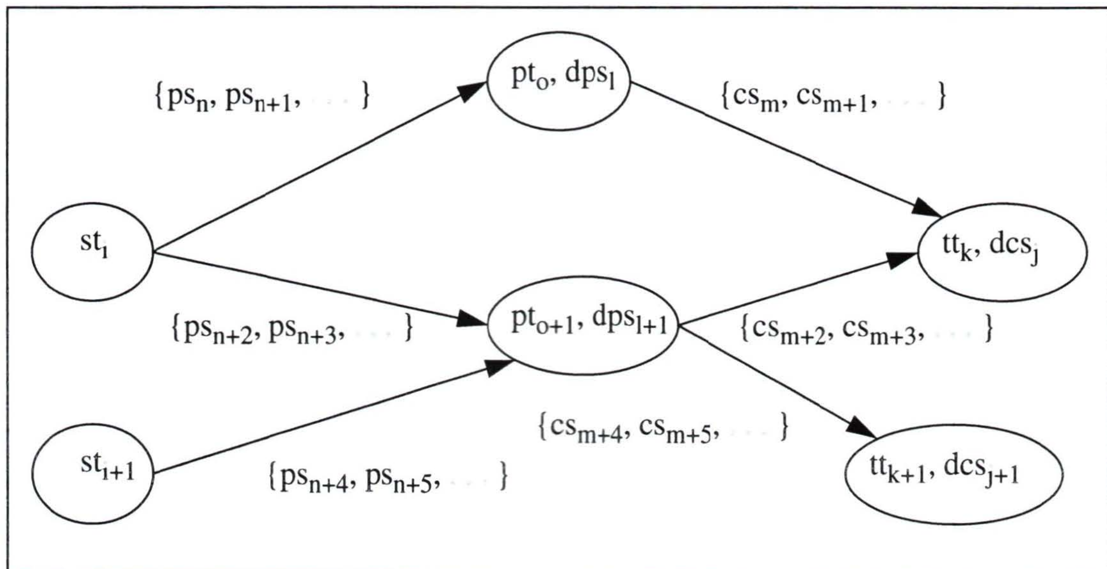


Figure 10. Refined version of Figure 9

match, the compiler would generate that particular change of state and produce its output more directly than processing the source text in the usual way.

3.3 Practical limitations on testing for equivalence

It is unrealistic to expect to be able to achieve the general goal outlined in the previous section since, in general, the cost of identifying and duplicating the effect of a fragment of text is going to be more than processing the text in the conventional manner. Furthermore, it is fairly obvious that every program fragment is not going to recur frequently, and also there is the question of identifying the largest possible reusable fragment. As a result we restrict ourselves to only attempt to duplicate the effect of text contained in header files. We expect to get a lot of leverage out of only considering header files for two reasons. Firstly, the very fact that the text has been put into a header file is a signal that it is intended to be widely used throughout the software system. Secondly, searching for a previous use of the header is extremely efficient since only file names and modification times have to be compared rather than the entire text of the header itself.

3.3.1 Equivalent preprocessing states

Having identified the inclusion of a header that has been previously processed, we proceed to determine if the preprocessing state is such that the effect of preprocessing will be the same as the last time the header was processed. As stated in the previous chapter, Litman [14] identifies preprocessing states as inconsistent in the case where a referenced identifier in the original state does not have the same definition in the new state or the new state contains a definition for an identifier appearing in the header that was not defined in the original state. This algorithm is conservative since it may miss opportunities to reuse precompiled header files of the following form:

```
#define SomeType int
extern SomeType foo;
```

This algorithm would be sensitive to the initial definitions for both SomeType and foo but this header guarantees a definition for SomeType so the actual output of the preprocessor

only depends on the initial definition of foo.

Onodera [16] presents a more aggressive algorithm which is less sensitive to the preprocessing states. An identifier is described as free in a header file if and only if there is no definition for the identifier in file before it is used. Note that the set of free identifiers in a header is a function of preprocessing state as well; consider the following header file:

```
#ifdef A_FLAG
extern SomeType foo;
#else
extern OtherType bar;
#endif
```

For a state where `A_FLAG` is defined the free identifiers are {`A_FLAG`, `SomeType`, `foo`, `extern`}, if `A_FLAG` is not defined then the free identifiers are {`A_FLAG`, `OtherType`, `bar`, `extern`}. When a header file is first encountered an empty definition list is associated with the file. Each time a free identifier is encountered its current definition is added to the list. When the header file is subsequently encountered the definitions for each identifier on the list are compared with the current definitions and any differences indicate an inconsistency. Although this algorithm is more flexible it is still somewhat conservative, for example:

```
#ifndef SomeType
#define SomeType int
#endif
SomeType foo;
```

Here `SomeType` is free in the file but the output of the preprocessor will be the same in cases where `SomeType` is either undefined or it is defined to be int.

In order to describe more precisely the conditions that the above algorithms are checking we adopt some notation. Let L be the preprocessing state when the header file, h, was precompiled or last saved; let N be the preprocessing state at a point where the header file is to be included and we wish to determine whether or not the precompiled or saved representation can be reused. In the absence of any information about the header file, the only condition guaranteeing that the two states are consistent is $L = N$. Assuming a function

first which returns the set of first elements of a set of triples and letting I_h denote the set of identifiers appearing in header file, *h*, the condition sufficient for consistency equivalent to Litman's algorithm is

$$first\,(L \oplus N) \cap I_h = \varnothing \tag{1}$$

where $L \oplus N$ denotes the symmetric difference of the sets *L* and *N* and $first\,(S) = \{x | <x, y, z> \in S\}$ The condition expresses the idea that any identifier that had a definition in the last state must have the same definition in the new state and that any identifier that has a definition in the new state must have the same definition in the last state (if the identifier appears in the header) Letting F_h denote the set of free identifiers in the header with respect to state *L* the conditions sufficient for consistency equivalent to Onodera's algorithm are the same as in equation (1) above but substituting F_h for I_h

A proof that the above conditions are sufficient for consistency follows

Theorem: The inclusion of a header file, *h*, in two states *L* and *N* generates the same sequence of tokens provided the definitions of any free identifiers, with respect to *L*, are the same in both states

Proof:

Equation (1) above, with F_h substituted for I_h, is equivalent to the statement the definitions of any free identifiers are the same in both states The proof proceeds by structural induction on the header file

Case 1 1 The file is empty The inclusion of an empty header will yield no output, which is the same in both cases

Case 1 2 The file contains a single token which is not an identifier The preprocessing of a token which is not an identifier is unaffected by the state of the preprocessor and so the output is the same in both cases

Case 1 3 a) The file contains a single identifier *id* which has no definition in *L* Since *id* is the only thing in the file, it is certainly free with respect to *L*, so $F_h = \{\ id\ \}$ Since (1) is

satisfied then there is also no definition for *id* in *N* and so the output is id in both cases.

Case 1.3. b) The file contains a single identifier *id* which has a definition in *L* with no arguments. Since *id* is the only thing in the file, it is certainly free with respect to *L*, so $F_h = \{id\} \cup S$ where *S* is the set of identifiers encountered on recursively rescanning the replacement string. Since (1) is satisfied, there is a matching definition for all identifiers in F_h in *N* and so the output is the sequence of tokens that *id* is defined to be after rescanning in both cases.

Case 1.3. c) The file is of the form:

$$id(arg_1, arg_2, \ldots, arg_n)$$

where *id* has a definition in *L* with *n* arguments. The identifier, *id*, any identifiers appearing in any of the arguments and any identifiers formed by the ## operator (the ## operator joins its operands if one of the operands is a formal argument, e.g. `argX ## 123` would be replaced with `abc_marg123` if `abc_marg` was the actual parameter for argX) in the replacement sequence for *id* are all free with respect to *L*, so $F_h = \{id\} \cup A \cup S$ where *A* is the set of identifiers appearing in the arguments and *S* is the set of identifiers encountered on recursively rescanning the replacement string, including those formed by the ## operator. Since (1) is satisfied, there is a matching definition for all identifiers in F_h in *N* and so the output is replacement sequence of tokens that *id* is defined to be with the appropriate argument substitution and rescanning in both cases.

Case 1.4. The file contains a single directive of the form: `#define` *id token_sequence*. This directive generates no output so the output matches in both cases.

Case 1.5. The file contains a single directive of the form: `#undef` *id*. This directive generates no output so the output matches in both cases.

Case 1.6. The file contains a single directive of the form: `#include "h1"`. Assume the theorem is true for the text of *h1* alone. The effect of this directive is to replace itself with the text of the named file. If we define $F_h = F_{h1}$ then (1) will be satisfied if the directive is

replaced with the text of the named file and since we have assumed that the theorem is true for the text of the named file then the output will be the same in both cases.

Case 1.7. The file, *h*, is of the form:

```
#if constant-expr
h1
#else
h2
#endif
```

Assume the theorem is true for *h1* and *h2* alone. Let *E* be the set of identifiers appearing in *constant-expr*. It is certainly the case that $E \subseteq F_h$ and since (1) is satisfied each identifier will have matching definitions in both *L* and *N*, thus either *h1* is processed in both cases or *h2* is processed in both cases. If the value of the constant expression with respect to *L* is 1 then $F_h = F_{h1} \cup E$ and since (1) is satisfied for F_h it will certainly be satisfied for F_{h1} and since the theorem is true for *h1* the output will be the same in both cases. If the value of the constant expression with respect to *L* is 0 then $F_h = F_{h2} \cup E$ and (1) will certainly be satisfied for F_{h2} and since the theorem is true for *h2* the output will be the same in both cases. Note that the `#ifdef` and `#ifndef` forms of conditional compilation can be converted into the `#if` form using the `defined()` operator and so the theorem is valid for these forms as well.

Case 1.8. The file, h, is of the form:

```
#error token_sequence
```

or

```
#pragma token_sequence
```

The `#error` directive only generates a diagnostic message and no output to the compiler so the output text is the same in both cases. The `#pragma` directive causes implementation defined behaviour to occur. In our case, pragma directives are ignored so no output is generated and the output to the compiler is the same in both cases.

Case 2 The file, h, is of the form

```
h1
h2
```

Assume the theorem is true for $h1$ and $h2$ alone Obviously $F_{h1} \subseteq F_h$ and so (1) will certainly be satisfied for F_{h1} since it is satisfied for F_h thus the output of the processing of the part of h corresponding to $h1$ will be the same in both cases since the theorem is true for $h1$ In order to show that the output of the processing of the part of h corresponding to $h2$ will be the same in both cases we need to show that (1) will be satisfied for L', N', and F'_{h2} where these correspond to the states and the free identifiers after the processing of $h1$ The only reason that L', N', and F'_{h2} would differ from L, N, and F_{h2} would be if `#define` and `#undef` directives were encountered in the processing of $h1$ If a `#define` directive is encountered then the same definition is added to (or replaces an existing definition in) L and N and the identifier is (possibly) removed from F_{h2} and so (1) is still satisfied If an #undef directive is encountered the definition of the identifier is (possibly) removed from L and N and the identifier is (possibly) removed from F_{h2} and (1) will still be satisfied After $h1$ is completely processed L', N', and F'_{h2} will continue to satisfy (1) and thus the output of the processing of the part of h corresponding to $h2$ will be the same in both cases

The above proof is only valid for h and L for which the expansion of h with respect to L is finite In general it is possible that a particular header file will generate an infinite expansion under a certain state as a result of processing infinitely recursive `#include` directives or expanding a recursive macro These situations are of little practical interest since an infinite expansion could never be compiled anyway If we imagine a pointer moving through the input text as the proof proceeds we remark that the remaining input past the pointer may grow temporarily as result of encountering an `#include` directive or expanding a macro However, ultimately the construct immediately following pointer will be an instance of base case 1 2, 1 3a, 1 4, 1 5, 1 7, or 1 8 and input can be consumed and the pointer advanced

3.3.2 Equivalent compilation states

Having established that the preprocessor would produce the same text for a given header for some pair of preprocessing states, we now consider the conditions on the state of the compiler proper required in order for the text of a preprocessed header to generate the same effect in two different states. In general we have the same requirement as for the preprocessor: any symbol with an exposed use in the header must have the same meaning in both states. In contrast to preprocessor where the meaning of “sameness” simply means that the sequence of tokens are the same, sameness in the compiler proper is more complicated. For example, in the declaration:

```
blim avar[7*foo];
```

there are two exposed uses of symbols `blim` and `foo`. The attributes of both of these symbols must be the same in both states. Dealing with `foo` is easy; it must be an enumeration literal with the same value in both states. The case of `blim` is a little more complicated; it must be a `typedef` name and if it names a structure or union type then all the symbols that are fields of the type must have identical attributes as well, and if any of those fields are of structure or union type then their field symbols must have the same attributes, and so on. Checking the general condition could be complicated and expensive. Instead we take an approach that is more restrictive, dictated primarily by the implementation we have chosen. We say the effect of header is the same provided any symbols with exposed uses are declared in other header files and that set of header files has been previously included in both compilations. The problem of ensuring that the headers have been included in the correct order is dealt with implicitly by testing the condition for each header as it is included, e.g. if A needs B and C and B needs C and we are considering the inclusion of A, and B and C have been included then C has been included before B because the test would have been performed for B when it was included. Under this method, we lose the ability to reuse headers which may have free identifiers defined in the body of the compilation unit rather than another header however we do not expect this situation to occur very often in practice.

4 Implementation

We chose to modify an existing C preprocessor and compiler rather than implement a compilation server from scratch as in the case of the COB compiler. The overhead of reimplementing a lexical analyzer, parser, semantic analyzer, and code generator seems unnecessary given that several free C compilers of acceptable quality are already widely available. Furthermore, the experience gained from modifying a compiler for a purpose which presumably was never considered in the original design may be instructive for anyone considering similar modifications to other compilers.

A natural selection would have been the GNU C compiler. Given that we were unfamiliar with the implementation details of any freely available C compilers, this choice was rejected due to its large size and complexity. In the final analysis, we settled on the C preprocessor that comes distributed with the X window system distribution and the lcc compiler [8][1]. Using thousands of lines of code (KLOC) as a measure of source code complexity, the GNU compiler consists of roughly 322 KLOC in the preprocessor and front end of the compiler, in contrast the selected preprocessor and front end consist of roughly 16 KLOC. The GNU compiler's size results from the fact that the front end processes three languages: C, C++, and Objective-C, also the compiler has a large number of compilation flags for setting many different modes of operation.

---

1. The technical report describing the compiler has been significantly expanded and will be published in book format in *A Retargetable C Compiler: Design and Implementation* [7].

4.1 Overview

The current structure and interaction of the existing preprocessor and compiler are shown in Figure 11. The preprocessor is given a primary input file and it may read other files as a

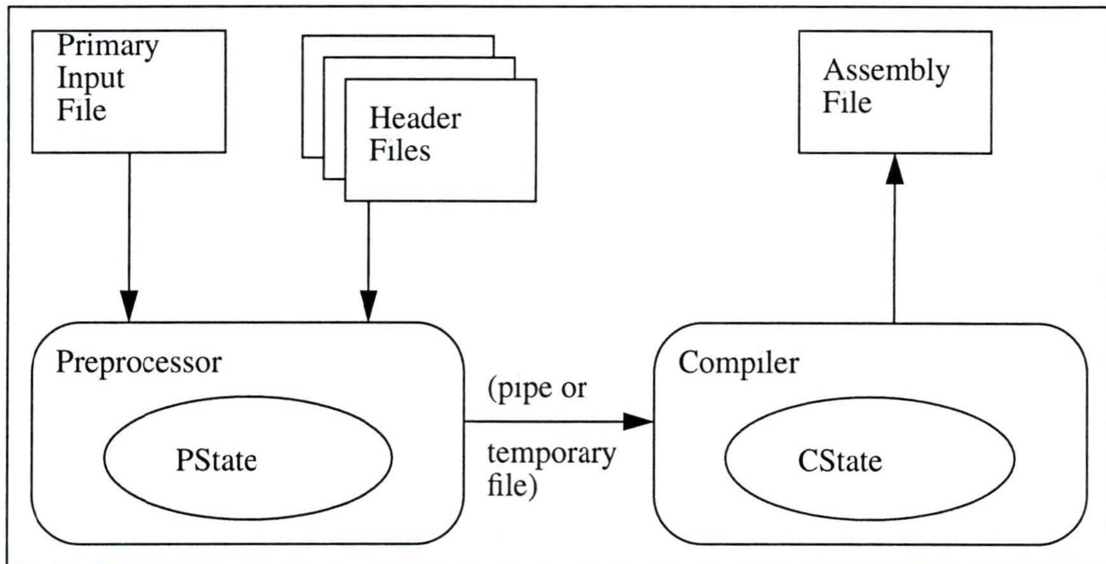


Figure 11. Original compiler organization

result of `#include` directives. As it processes text its state may be updated and it may generate output text. The preprocessed text is passed to the compiler either through a temporary file or pipe. The compiler processes text generated by the preprocessor updating its state or generating assembly code. The preprocessor terminates when it reaches the end of the primary input file. The compiler terminates when it reaches the end of the temporary file or the pipe is closed.

The new structure of our implementation is shown in Figure 12. In addition to state, the preprocessor maintains internal representations (IRs) of header files, the same is true of the compiler. The preprocessor and compiler communicate by a pipe. Additionally, the compiler maintains the preprocessed text of headers in files for the event that it is unable to use its IR of a header. Both processes are long-lived, processing an indefinite number of compilation units, and must be terminated explicitly.

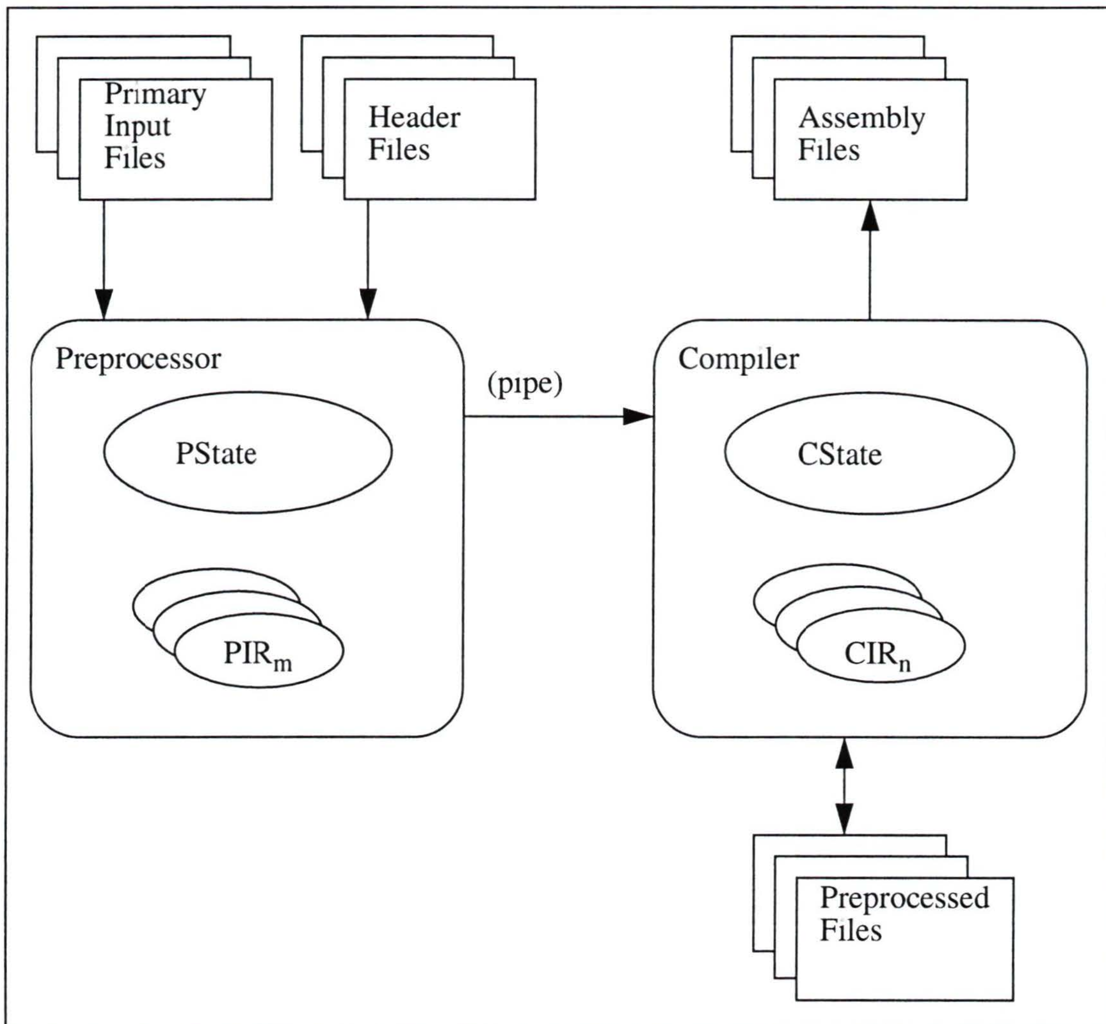


Figure 12. New compiler organization

The remainder of this chapter describes the details of maintaining and reusing IRs in the preprocessor and compiler proper.

4.2 A preprocessing server based on cpp

Interestingly, the X window system comes with the source for a C preprocessor. This is because the preprocessors that come bundled with some operating systems cannot cope with the huge number of macro definitions that must be processed when including the header files for the X window system. Although the implementation is easy to understand,

it is not particularly efficient. The preprocessor was compared against others and the relative results are shown in Table 1. An execution profile revealed that the preprocessor

X cpp/lcc	cc	acc	gcc
1.00	0.72	0.74	0.80

Table 1: Relative performance of preprocessors

spends 32 percent of its time in a function which returns the next significant character on the input stream. Performance probably could be improved significantly by basing the preprocessor on a function that returns a token at a time rather than a character at a time.

Three major changes were required for transforming the conventional preprocessor into a preprocessing server. Mechanisms are required for: creating and storing the IR of a header file, establishing when the IR of a header file can be reused, and finally, using the IR to update the state of the preprocessor. The representation of the state of the preprocessor is key to all three tasks and hence we examine it first.

4.2.1 Preprocessing state representation

As discussed in the previous chapter, the preprocessing state is defined by a set of macro definitions. Since the source text of programs typically contains many identifier tokens, the set is implemented by a hash table where collisions are resolved through chaining in order to determine quickly whether or not identifiers have a macro replacement. The hash function used by the original implementation sums the ASCII values which we call the full sum (FS). The value of the last few bits of the FS are used as the index into the hash table. The FS is stored with each identifier in the table in order to speed up comparisons along the hash chain; only the FS values of identifiers are compared until a match is found and then the more expensive string comparison is done.

The only change we made to this scheme was to order the chains primarily by FS value and lexicographically for elements with the same FS value. This has the benefit that lookup and insertion are still linear based on the load factor of the table but set compari-

son, union, and intersection can be performed much more efficiently than for the unordered case. If we have two sets implemented by unordered hash tables whose load factors are α and β then the worst and expected case performance for the set union operation is $O(\alpha\beta)$; whereas for ordered tables the worst and expected case performance is $O(\alpha+\beta)$.

4.2.2 Internal representation of header files

The IR of a header file contains three pieces of information: the set of files upon which the IR depends, the set of exposed identifiers along with their definitions when the IR was created, and the set of changes to state of the preprocessor the header file represents.

The set of files upon which the IR depends along with their update times is required in order to establish whether or not the IR is up to date with respect to the source text. If the text of one of the dependent files has changed then, at least without any other information, the IR must be discarded. Since this set is relatively small, it is represented simply by a linked list. The list is constructed when the IR is created by adding an element to the list each time an `#include` directive is encountered when processing the header and storing the name of the file and the time of modification.

Having found an IR for which the set of dependent files are unchanged since the IR was created, the next step is to check the set of exposed identifier definitions against the current state of as described in the previous chapter. If no IR has a consistent set of exposed identifier definitions then another IR is created. The set of exposed identifier definitions is represented by a hash table with the same size and hash function as the preprocessor state table. The table is constructed when the IR is created by inserting an element into the table each time a new exposed identifier is encountered in the header and storing the identifier's current definition.

Once an IR with a consistent set of exposed identifier definitions is found, the final step is to apply the set of changes to the current preprocessor state. The set of changes is also represented by a hash table with the same size and function as the preprocessor state table. The table is constructed when the IR is created by inserting an element into the table each

time a `#define` or `#undef` directive is encountered in the header. If there is already an entry in the table for an identifier when it is inserted, then the definition is overwritten since, in terms of updating the state of the preprocessor, only the final definition for an identifier really matters.

4.2.3 Identifying consistent states and updating state

Since we have the convention that all hash tables are the same size, they use the same hash function, and all hash chains are kept in sorted order then we can implement the state consistency test efficiently. In general, each exposed identifier in a header file has to be looked up in the current preprocessor state to ensure that its definition (or lack of one) matches the definition for the identifier when the IR was created. Given that both sets are represented with similar tables, the test can be implemented simply by traversing corresponding hash chains in tandem, in a similar manner to a merging operation. The updating of the preprocessing state can be carried out in a similar manner, inserting, deleting, and updating elements along each hash chain as appropriate. In Figure 13[1], an IR along with the state of the preprocessor before and after the use of the IR is shown.

4.2.4 Memory management

It should be obvious that a preprocessing server cannot go on creating IRs indefinitely without deleting other IRs. Eventually a point will be reached where it becomes more expensive to find a suitable IR rather than to process the text of the header file in the normal way. Also, we would ideally like a situation where a majority of IRs are in memory instead of paged out to disk. Instead of trying to get an accurate estimate of how much memory each IR takes up and adjusting the number of IRs retained accordingly, our implementation simply retains a fixed number of IRs. This scheme may leave of lot of memory unused in the case where all the IRs are small, or lead to a lot of paging if all the IRs are large, but we expect it to work well in the normal case. When the server has

---

1. The exposed definition set and changed set are actually represented by hash tables but are illustrated in the figure as linked lists for clarity. In the remainder of this thesis hash tables will be illustrated as linked lists since it is the list structure of the hash chains that is usually significant.

reached its limit on IRs and needs to create a new one, an existing one is discarded based on a least recently (or frequently) used strategy. The selection strategy may be tempered with size strategey as well, choosing to discard larger IRs instead of smaller ones for example.

An IR normally consists of a large number of small dynamically allocated objects. In our situation, we have a set of objects that we want to allocate space for individually and deallocate as a group. The conventional C library functions `malloc()` and `free()` do not provide this functionality directly. Instead we borrow the memory allocation scheme provided in the original compiler proper [10]. Essentially, `malloc()` is used to dynamically allocate large segments of memory as needed and objects are allocated contiguously within each large block. When the objects, as a group, are no longer needed the large blocks are freed instead of freeing up each object individually. In our implementation, each IR has its own list of large blocks (*arenas*) and when the IR is discarded each arena on the list is freed. This memory management scheme distinguishes our implementation

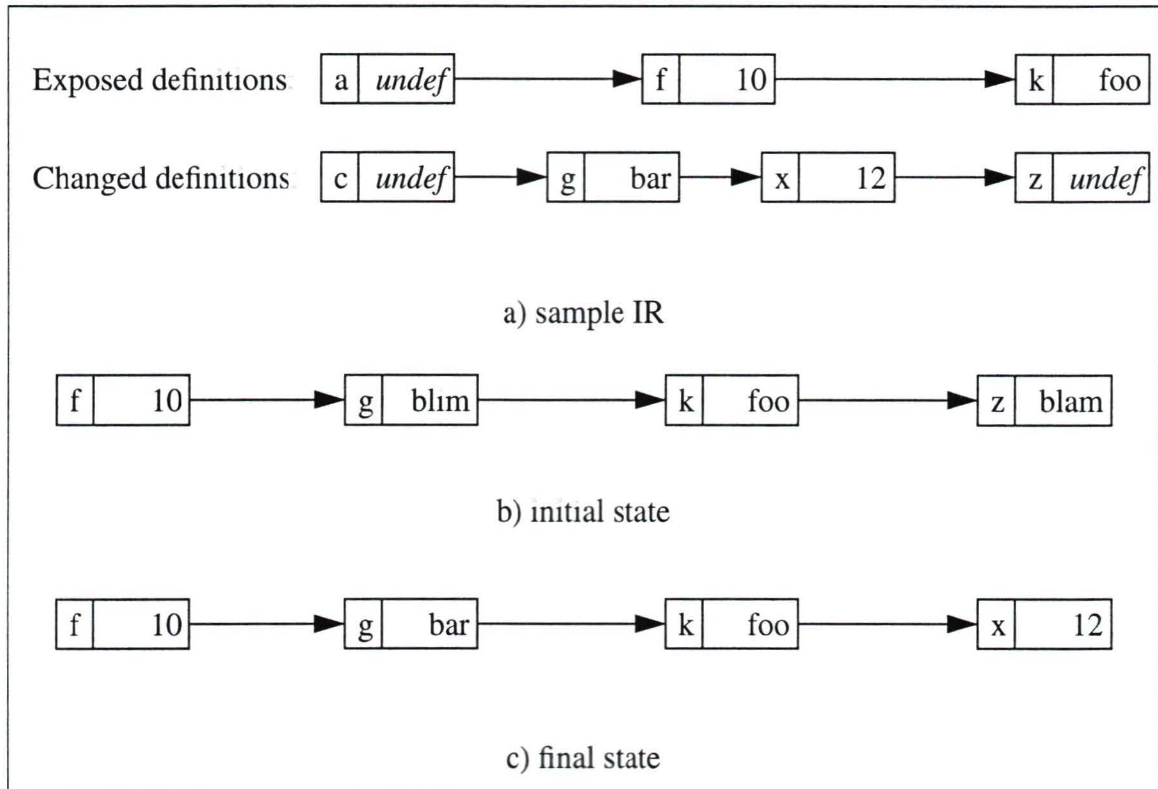


Figure 13. Example use of an IR

from Onodera's which relies on garbage collection in the runtime environment to reclaim storage from discarded objects. Our observation is that source programs have structure, grouping many small objects together which get discarded all at once so an explicit deallocation scheme is simpler to implement as well as more efficient.

4.3 A compilation server based on lcc

Lcc is a small fast retargetable ANSI C compiler. Its speed stems from an efficient front end which includes a hand written lexical analyzer and recursive descent parser. Its small size precludes a global optimization phase, however the inclusion of important local code transformations means it normally generates reasonably good code. The compiler is implemented so efficiently that it overcomes the slowness of the preprocessor such that the relative combined performance is better than compilers listed in Table 2.

X cpp/lcc	cc	acc	gcc
1.00	1.08	1.28	1.79

Table 2: Relative performance of combined preprocessors/compilers

Part of the attraction of transforming lcc into a compilation server is the arena method used for storage management described briefly in the previous section. The original compiler made use of two arena lists, one for objects which persist for the duration of the translation of a compilation unit and one for objects local to a function definition. The arenas for local objects are reused for each function defined in the translation unit. In the modified compiler we extend this by having arena lists which persist for the duration of the server process, for the duration of one translation unit, and for the duration of one function definition. Additionally, there is an arena list for each IR of a header file.

Like the preprocessor, three major additions are required to transform the compiler from a short lived process that translates a single compilation unit to a server which translates several. Procedures are required for creating and storing the IR of a header, identifying when the IR of a header can be reused, and updating the compilation state. All these pro-

cedures depend somewhat on the representation of compilation state and so we describe that first

4.3.1 Compilation state representation

The compilation state is defined by a set of symbol declarations Like the preprocessor, since the source text contains many symbols, the set is implemented by a hash table in order to locate symbol declarations quickly Collisions in the table are resolved through chaining Unlike the preprocessor, the elements of this set are much more sophisticated Whereas a macro identifier simply has zero or more arguments and a replacement sequence of tokens, a symbol in the compiler has attributes such as type, name space, and scope, furthermore, a symbol may depend directly or indirectly on the declaration of other symbols A further difference is that elements of the set are hashed based on a pointer to a string representing the identifier rather than the string itself, this creates some difficulties which are described below The symbol table organization was not altered significantly except for the addition of attributes for each symbol which will be described as required below

4.3.2 Internal representation of header files

The IR of a header file contains two distinct pieces of information the set of IRs upon which the IR depends and the set of symbols which the IR represents

The set of IRs upon which the IR depends is used in order to establish whether or not the IR can be used in the current compilation For each compilation we record the set of IRs in use, if this set is a superset of the IRs required then we can use this IR and add to the set of IRs in use, otherwise we process the preprocessed text of the header file directly Since this set is relatively small and there is a fixed number of possible IRs available, the set is implemented as a bit vector, this means the superset test can be carried out as a binary AND operation The set is constructed when the IR is created by recording the IR from which each referenced symbol comes in the bit vector

If all the dependent IRs of an IR are in use, then the next step is to update the compilation state The set of symbols which the IR represents is a hash table with the same size and format of the table representing the compilation state The table is constructed when the IR is created by initially saving the pointers to the last elements in the hash table representing the compilation state When the end of the text corresponding to the header file is reached, the pointers to the last elements in the hash table are saved These two sets of saved pointers completely define all the symbols declared in the header Note that the symbol table does not contain any representation for initializers of a variable or function bodies so any declarations in a header file which contain a constant initializer or function body are lost and render the IR unusable Furthermore, a header file can contain arbitrary text and so may contain fragments of declarations, in this case the IR is unusable as well In either of these two cases, the preprocessor may have a valid IR of a header whereas the compiler may not have a corresponding IR

In the event that not all the IRs are in use that are depended on by an IR, it may be possible to create an additional IR for the preprocessed header text just like the preprocessor may create several IRs for the same unpreprocessed header files We expect the situation of having several preprocessor IRs of an unpreprocessed header to be infrequent and the case of having several compiler IRs of a preprocessed header even more infrequent so we do not even bother at the level of the compiler proper Thus the compiler contains at most one IR for each preprocessor IR

4.4 Reusing IRs in the compiler proper

Since the IR has a hash table of the same size and format of the table representing compilation state, we can reuse an IR simply by linking in the corresponding hash chains into the main table The entire concept of retaining and reusing these symbol table fragments is illustrated in Figure 14 In part a) the symbol table in effect at the end of processing a compilation unit which included three header files is illustrated Part b) shows what is retained between compilations Part c) shows the reuse of two of the IRs in a subsequent compila-

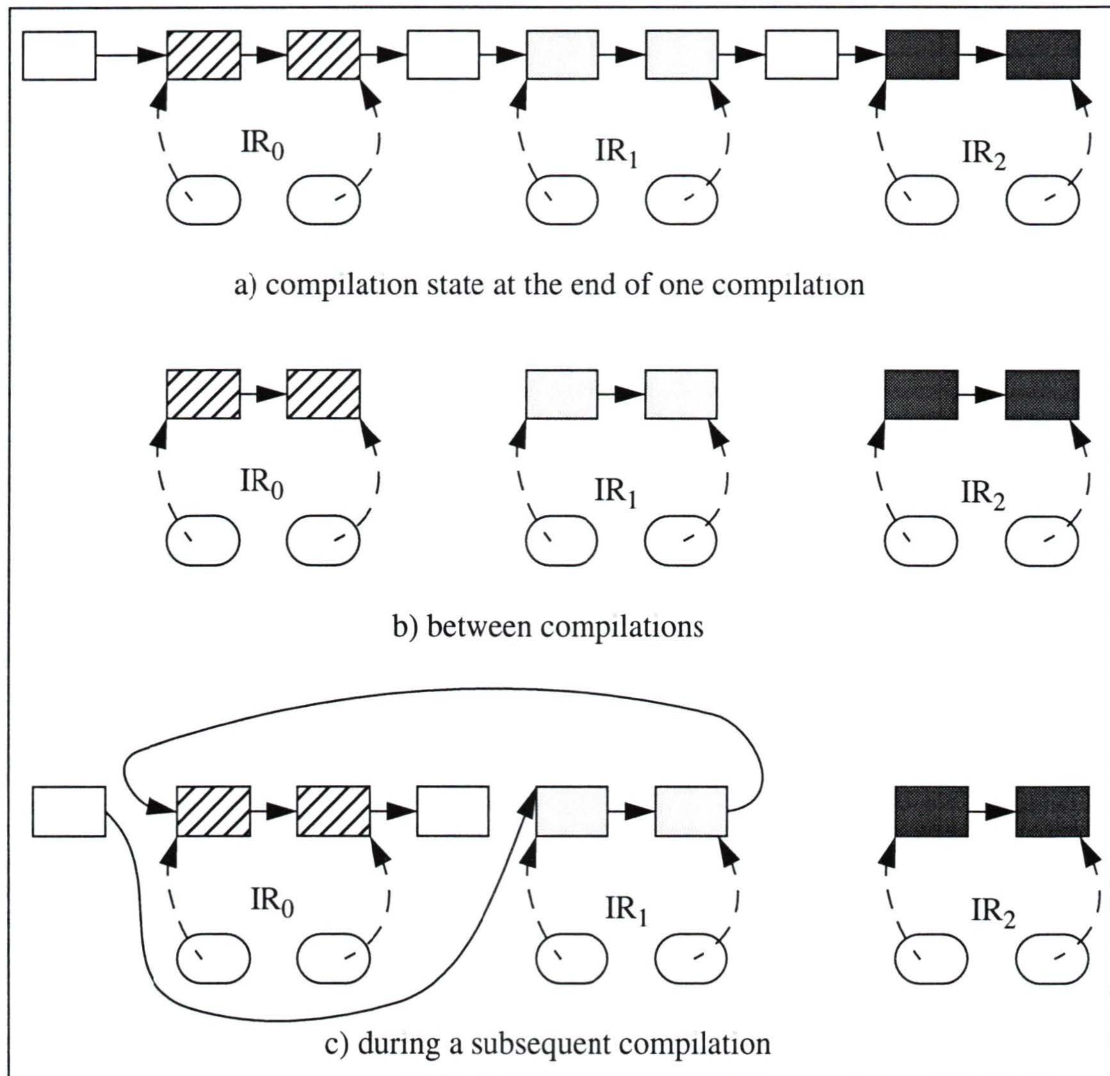


Figure 14. Reuse of compiler IRs

tion unit. Whenever an IR is reused we simply flag all symbols in the IR as reused and defer any further analysis until a symbol is actually referenced elsewhere in the program text. This has the benefit that we do very little work for unreferenced symbols and we expect the majority of symbols declared in header file to go unreferenced in a particular compilation unit.

It is the repetitive movement of large number symbols in and out of the symbol which distinguishes our work from Onodera's. Onodera's implementation restricts itself to retaining only interface files which get implicitly included and so any symbols declared in the inter-

face file can be left in the symbol table for the life of the compilation server or at least until the text of the interface file changes. In contrast, in C, header files are explicitly included and so the symbols declared therein can only be present in the symbol table if the file has be explicitly named.

4.4.1 Deferred semantic analysis of declarations - laziness

A declaration in C typically declares one or more symbols and may reference zero or more other symbols. For example:

```
struct blim blam[10], zip[foo+bar*3];
```

declares `blam` and `zip` and references `foo` and `bar`. There are additional constraints: `blim` must have been declared previously or eventually as a structure tag, `foo` and `bar` must be enumeration constants, `blam` and `zip` must not have been declared previously (or if the declaration is in the global scope then they can have multiple declarations as long as the types are compatible). We ensure that each referenced symbol in an IR of a header file has at least one definition when the IR is reused through the use of the dependency information. We do not ensure that there is exactly one definition (or multiple compatible definitions) for referenced symbols or compatible definitions for declared symbols until a declared symbol is actually referenced. This lazy strategy is expected to save time in the case that compilation unit makes little use of symbols declared in header files. The primary drawback is that the server may accept non-conforming programs however we consider the non-conformance "harmless" in the sense that multiply (or incompatibly) defined symbols are never referenced, i.e. for example, the incompatible declarations:

```
struct foo {
      char a;
};
struct foo {
      double b;
      int c;
}
```

are harmless provided the structure tag `foo` is never used later on in the compilation unit. The remainder of this section describes in more detail some of the issues surrounding this

lazy analysis strategy.

Type equality is a problem because, like strings, types are frequently compared by comparing pointers. If a data structure represents a type and two symbols have their types represented by pointers to a type data structure, then their types are deemed to be equal if their type pointers are equal. This test breaks down when we start reusing symbols generated by separate compilations especially where structure and union types are concerned. Consider the following program fragment:

```
struct foo a;
struct foo b;
```

The first declaration declares `foo` as a structure tag and creates a structure type and declares `a` to be a variable of that structure type. The second declaration looks up the declaration for `foo` and declares `b` to be a variable of the same structure type. The symbol table entries for `a` and `b` both have the same value in the type pointer field and so would be deemed to be the same type. Now consider the program fragment in Figure 15. A conven-

```
/* a.h */
struct foo a;

/* b.h */
struct foo b;

/* t1.c */
#include "a.h"

/* t2.c */
#include "b.h"

/* t3.c */
#include "a.h"
#include "b.h"
a = b;
```

Figure 15. Type equality example fragment

tional compilation of `t3.c` would have the same effect as described above and the assignment would be valid because `a` and `b` would have identical type pointers. Consider instead the compilation of `t1.c`, `t2.c`, and `t3.c` in that order by a compilation server. Compiling `t1.c` creates a structure type for `foo`. Compiling `t2.c` creates *another* structure type for *foo*. The conventional compiler's symbol table only ever has one entry for a particular symbol. The compilation server may have multiple entries for the same symbol

when IRs are taken from different previous compilations. In this example when `t3.c` is compiled there would be two entries for `foo` in the symbol table, both naming distinct types as far as pointers are concerned, hence the types of `a` and `b` would be deemed to be different because their type pointers point to two different structures even though the types are really the same. We correct for this by modifying the type equality operation. If two type pointers do not match but they are both structure/union types and their tags are the same then the types are equal.

A second problem is incomplete structure and union types. Depending on where a header is included, a structure or union type may be complete or incomplete and the type of reused symbols may have to be adjusted accordingly. We accomplish this by storing with each symbol its type when the end of its header was reached along with its current type. As a concrete example, consider the program fragment in Figure 16 and compiling `t4.c`

```
/* c.h */
struct bar *a;

/* t4.c */
#include "c.h"
struct bar {
   int x;
   float y;
};
```

```
/* t5.c */
#include "c.h"
a->y = 3.14;
```

Figure 16. Incomplete type example #1

followed by `t5.c`. A conventional compiler would detect the error in `t5.c` when referencing the `y` field of `a` since the `bar` record type is incomplete. The compilation server has to take care to save the incomplete type of a when it finishes processing `c.h` in `t4.c` and reset the type to the incomplete state when it reuses the symbols of `c.h` in `t5.c`. The correct action also has to be taken in the converse case, completing types for reused symbols where the type is completed earlier than the declaration of the variable. This is dem-

```
/* t6.c */                    /* t7.c */
#include "c.h"                struct bar {
                                 double blim;
                                 double blam;
                              };
                              #include "c.h"
```

Figure 17. Incomplete type example #2

onstrated in Figure 17 where `t6.c` is compiled before `t7.c`. Here the type for `a` has to be set to the completed type for the structure.

Another problem is dealing with duplicate declarations. In the conventional compiler each distinct object in the program has a unique entry in the symbol table. In the compilation server this may or may not be the case depending on the symbol in question. Functions and global variables can be declared many times provided that their types are compatible. Other objects, such as enumeration constants, can only be declared once. When a reused object is first used, the hash chain is searched for objects of the same name to ensure there is only one entry or if there is more than one entry that their types are compatible as well as unifying the types. We also store with a symbol a list of symbols which its declaration depended on and these symbols must be tested for validity as well.

A final issue surrounding this lazy strategy is that it is only applied to symbols contained in IRs. When the text of declaration is processed in the usual way, the symbol table is checked immediately for conflicting declarations. This has the unfortunate drawback that the order in which compilation units are compiled may have an effect on whether the compilation succeeds or fails depending upon which IRs are present in the compilation server. In Figure 18 a program fragment is shown which is compiled satisfactorily by the compilation server as long as `t2.c` is compiled before `t3.c`. If `t3.c` is compiled before `t2.c` then the error of the multiple declaration of the typedef name `X` is revealed because the text of both header files is processed.

```
/* h1.h */
typdef int X;
extern double a;

/* h2.h */
typedef double X;
extern char b;

/* t1.c */
#include "h1.h"

/* t2.c */
#include "h2.h"

/* t3.c */
#include "h1.h"
#include "h2.h"
void f(void) {
   a = 3.14;
   b = 'x';
}
```

Figure 18. Lazy order example

This apparent inconsistency is irritating at worst and although there was no perceivble benefit (or detriment) of the lazy strategy for compilation units in our test suite, we were able to construct compilation units for which the time savings were significant. We would classify compilation units that would benefit from this scheme as having a large overall number of declarations and where the ratio of declarations to executable statements is high. Although it is difficult to imagine hand written code satisfying this criteria, automatically generated programs might. The time savings is essentially proportional to the load factor for the hash table used to implement the symbol table. That is, the cost of checking each declaration basically involves searching for duplicate declarations of a symbol along the hash chain it is stored on, the cost of this operation depends how long the hash chain is, i.e. the load factor of the hash table.

4.4.2 Memory management

Memory management in the compiler is driven by the preprocessor. There is at most one IR in the compiler for each IR in the preprocessor. When the preprocessor discards an IR, either because a dependent text file has changed or because the space is needed to store a new IR, the corresponding IR in the compilation server must be discarded as well, furthermore, any IRs which depend on the discarded IR must be discarded as well.

The fact that identifiers are represented by pointers creates some difficulties. Every use of an identifier must have the same pointer value. A single identifier may be used in several different contexts in several different headers. The strings corresponding to identifiers are dynamically allocated in the arena style data structure and thus cannot be deallocated independently. Without any changes the string data area would grow indefinitely in size. One option would be to implement a reference counting algorithm and a separate dynamic allocation/deallocation scheme for strings so that the space for a string could be freed independently when no IR was left containing any references to a particular string. Instead we take a simpler, although less precise, approach discarding all strings and IRs when the string table becomes too full. This has the drawback that some IRs have to be recreated from the preprocessed source text.

5 Results

Having described our implementation, we now attempt to quantify its effectiveness. The measurements made in this chapter are based on the compilation of 253 compilation units taken from the preprocessor, the compiler, Emacs, Gnuplot, an image display/annotation utility, and two flight simulators. In order to provide some context for the analysis we compute the usage of declarations in each compilation unit.

5.1 Declaration usage

Our basic premise is that we can reduce compilation time by retaining the internal representation of header files provided that the header is included by several compilation units or the same compilation unit is compiled several times, and the usage of declarations contained in the header by the compilation unit is low. In order estimate the effectiveness of selective embedding, Adams et al. [1] use the Ratio of Use of Visibility (RUV) metric. For a given declaration, d, and a set of compilation units, the number of compilation units

where d is visible is counted, vis(d), and the number of units where d is actually used is counted, use(d). The RUV value for a given set, D, of declarations is computed as

$$RUV_D(D) = \frac{\sum_{d \in D} use(d)}{\sum_{d \in D} vis(d)}$$

In their analysis of a large program the RUV value for the set of all declarations appearing in specification units of the entire program was 0.2. We compute the same metric in a slightly different way. For each compilation unit, c, we count the number of identifiers declared in the compilation unit, decl(c), and the number of identifiers used in the compilation unit, iuse(c). The RUV value for a given set, C, of compilation units is computed as:

$$RUV_C(C) = \frac{\sum_{c \in C} iuse(c)}{\sum_{c \in C} decl(c)}$$

$RUV_D(D) = RUV_C(C)$ for corresponding sets of declarations, D, and compilation units, C, and so we simply refer the metric as RUV in the remainder of this section. In our analysis we consider all global identifiers appearing in a compilation unit rather than just the ones appearing in header files. We compute RUVs separately for macro identifiers and for all other identifiers. The results for macro identifiers are quite accurate since we only count an identifier in iuse(c) if the macro is actually expanded. The results for all other identifiers is somewhat more conservative since we count an identifier in iuse(c) even if it is only used in the declaration of other unused identifiers. As an example the following compilation unit has an RUV value of 0.5 because of the use of IntType in the declaration of foo even though the identifier foo is itself never used:

```
typedef int IntType;
IntType foo;
```

For our set of sample compilation units the RUV value for macro identifiers was 0.06 and for all other identifiers was 0.11. The distribution of RUV values for each compilation unit for macro identifiers and all other identifiers is shown in Figure 19 and Figure 20 respectively.

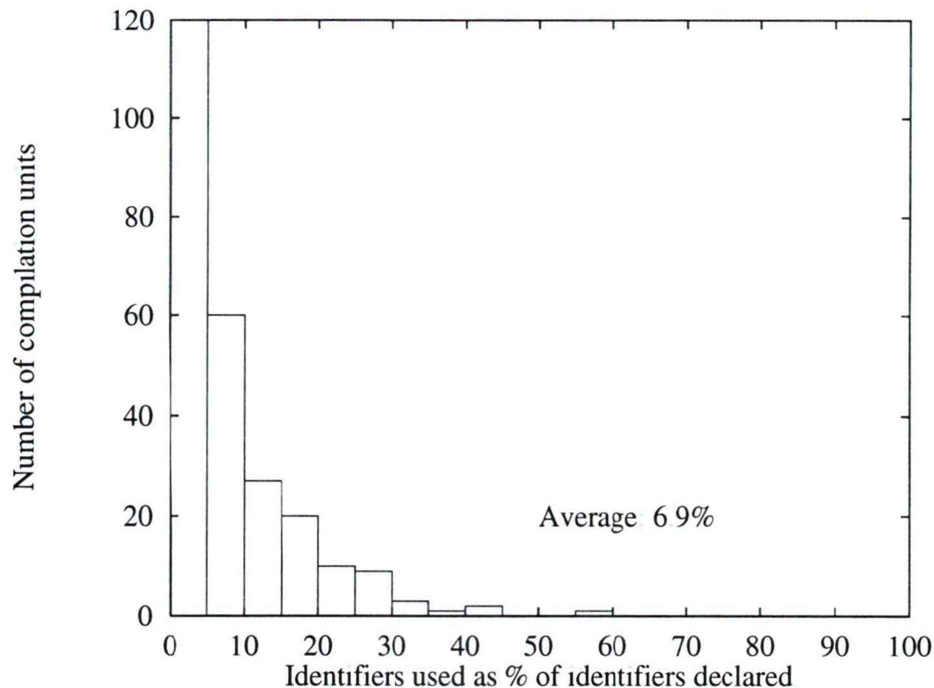


Figure 19. RUV distribution for macro identifiers

5.2 Methodology

The time and space requirements of the original compiler and preprocessor are used as the basis for most comparisons. The time it takes to compile a compilation unit is computed summing the system and user time values returned by the `getrusage()` function. Compilation time is broken down into time spent in the preprocessor and time spent in the compiler proper. Space requirements are assessed by measuring size of the data segment of the process which includes the static data area and the heap. The value is computed as `sbrk(0)-&etext`. We break down data segment size into size of the preprocessor and size of the compiler proper. We do not concern ourselves with the size of the text segment

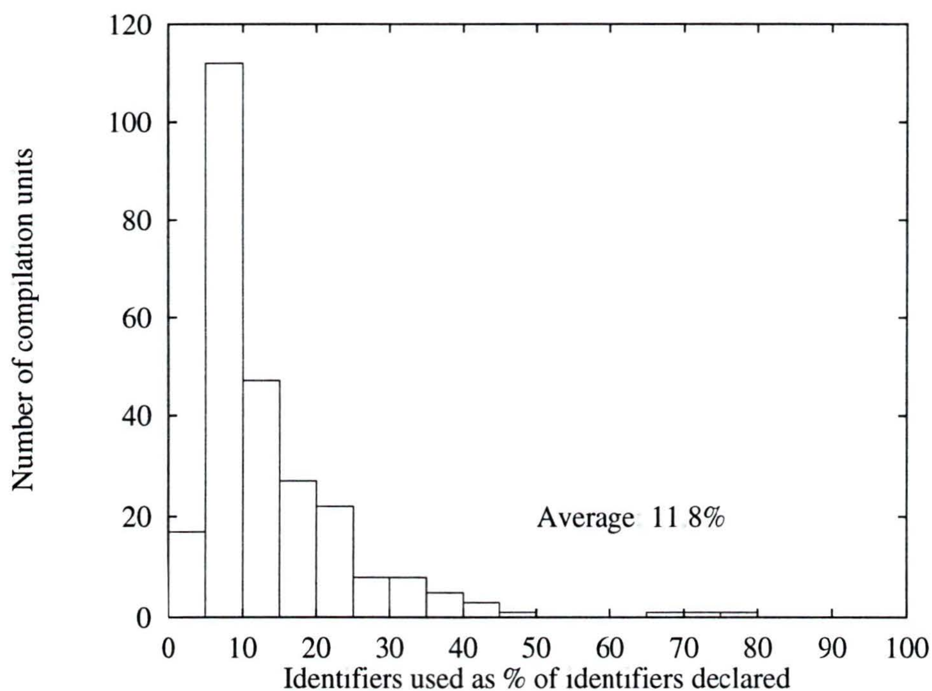


Figure 20. RUV distribution for all other identifiers

since its change in size is fixed and modest. The effect of our changes on stack size is also insignificant and is ignored as well.

5.3 Measurements

We measured the performance of the modified compiler in three situations. In the first case we looked at the situation where the server has no IRs of header files which we consider to be the worst case scenario. Secondly, we look at the case where the largest number possible of IRs are available for a compilation unit; this is the best possible case and we would consider it to be typical of the repetitive compilations executed in an edit-compile-debug cycle. The final situation involves the massive compilation of several related compilation units comprising a program or a library.

5.3.1 Initial compilation

The measurements for this situation were made as follows for each compilation unit in the test set, start the compilation server, compile the compilation unit, and record the compilation time and resulting size of the data segment

The distribution of preprocessing times (relative to the amount time spent in the original un modified preprocessor) is shown in Figure 21 The distribution of relative compilation

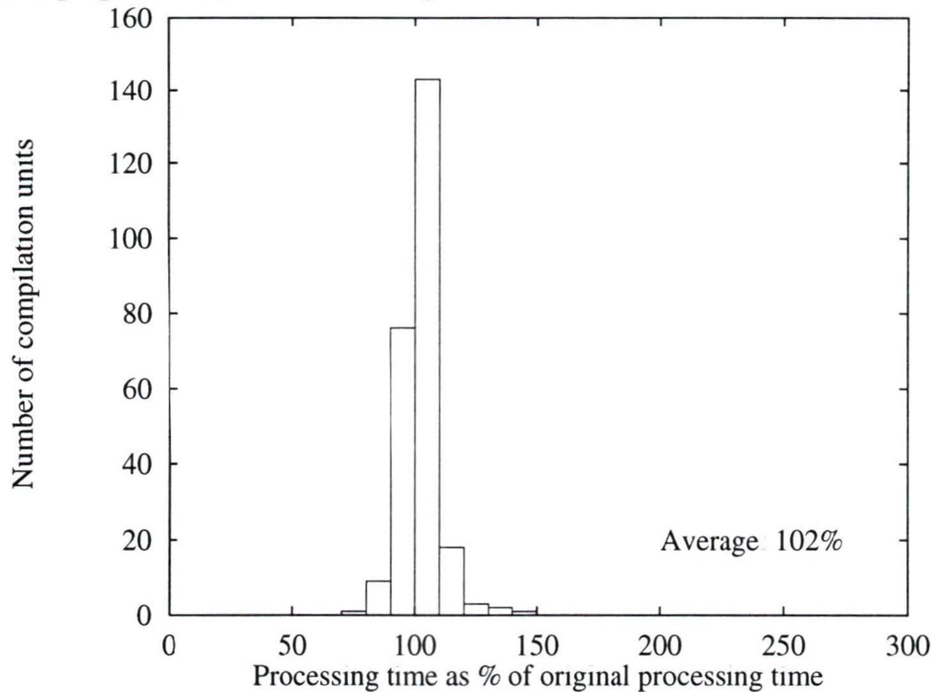


Figure 21. Distribution of relative initial preprocessing times

times is shown in Figure 22 The distribution of relative combined preprocessing and compilation times is shown in Figure 23 The graphs show that the overhead of constructing the IRs of header files is quite modest While the time penalty is small the space penalty is another matter In Figure 24 and Figure 25 the distribution of relative increases in

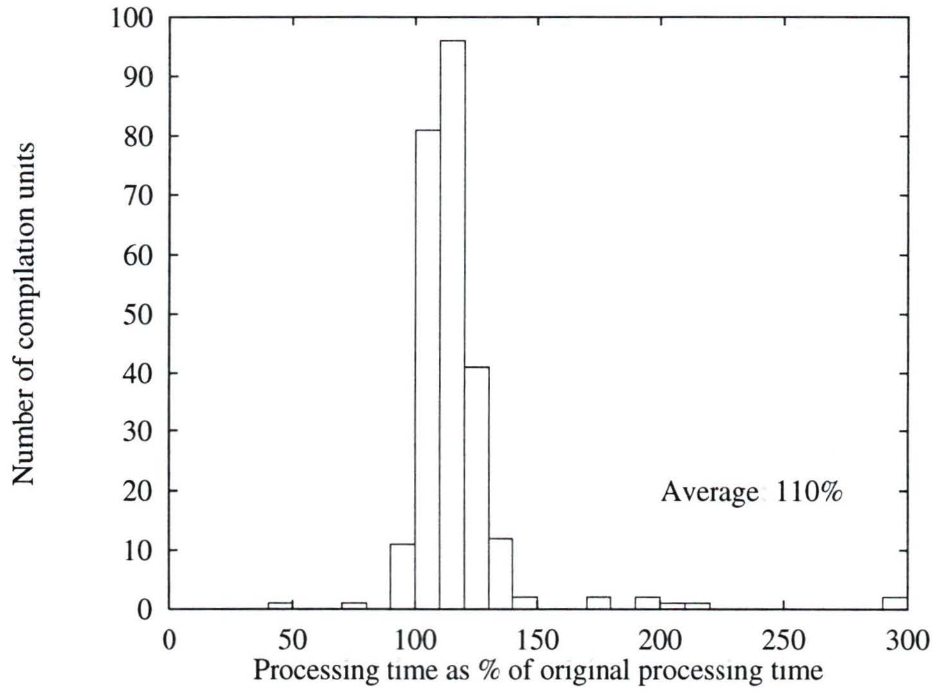


Figure 22. Distribution of relative initial compilation times

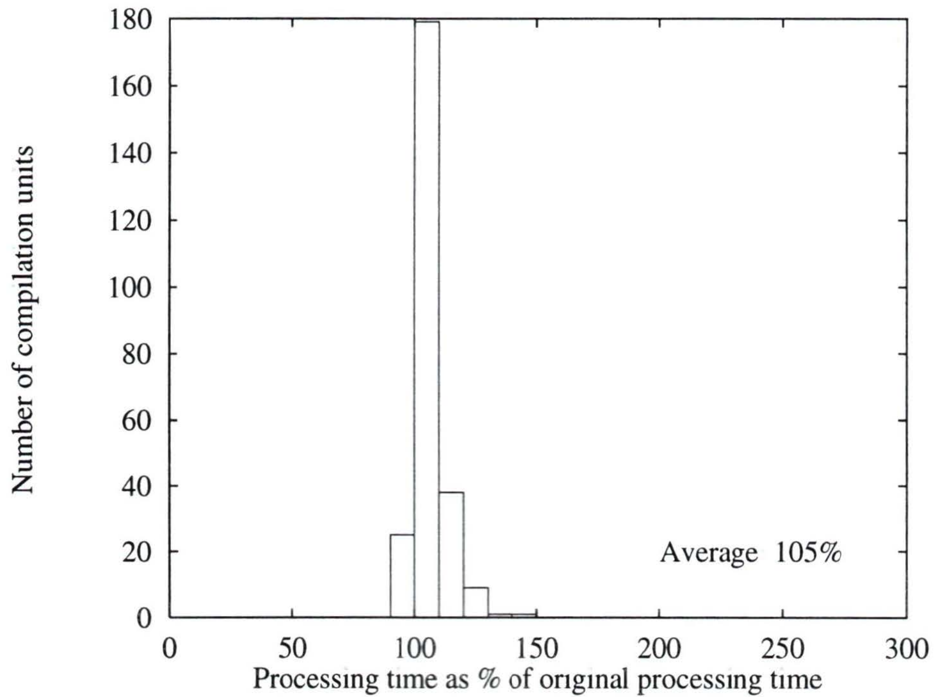


Figure 23. Distribution of relative initial overall times

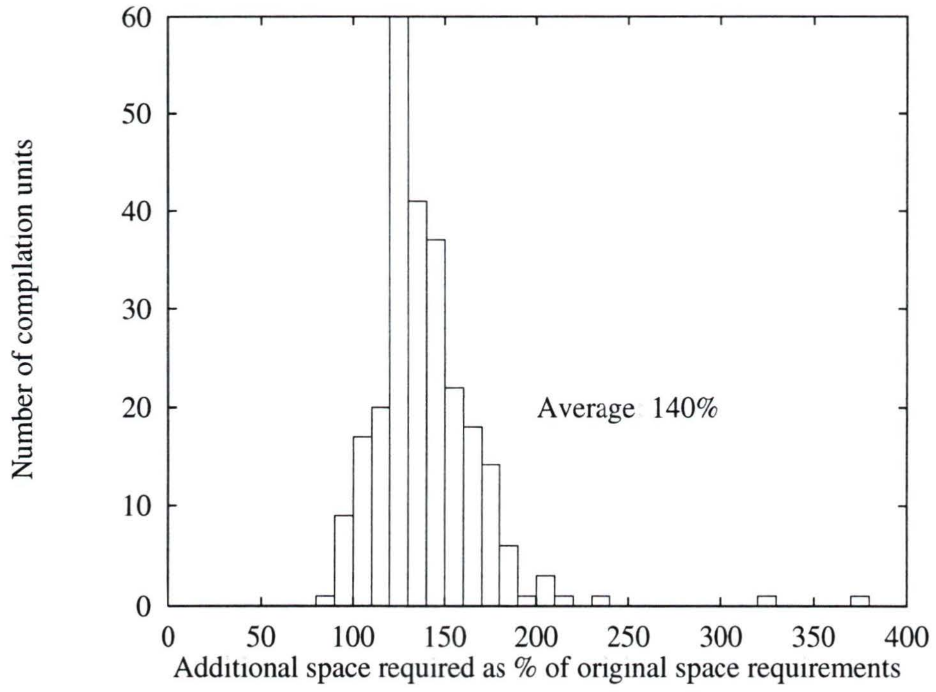


Figure 24. Relative increases in preprocessor data segment size

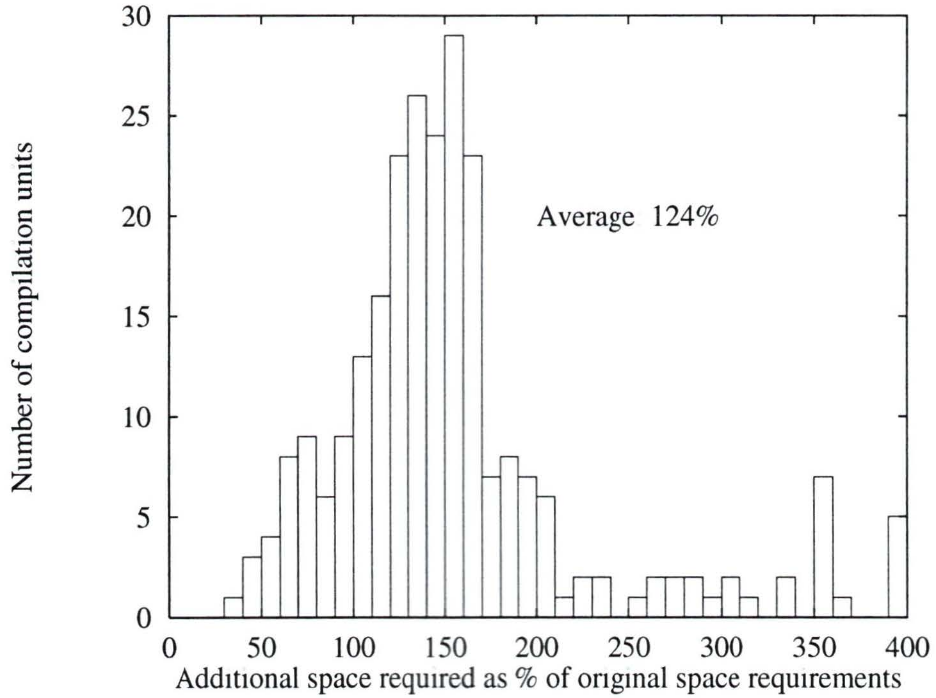


Figure 25. Relative increases in compiler data segment size

the size of the data segment is shown for the preprocessor and compiler respectively. In Table 3 the average absolute and relative data segment sizes are given.

Program		Absolute size (bytes)	Relative size
Preprocessor	Conventional	140135	1.00
	Modified	335959	2.40
Compiler	Conventional	342994	1.00
	Modified	768971	2.24
Combined	Conventional	483129	1.00
	Modified	1104930	2.29

Table 3: Data segment sizes

5.3.2 Repetitive compilation

The measurements for this situation were made as follows: for each compilation unit, start the compilation server, compile the compilation unit twice in succession, and record the compilation time of the second compilation. The sizes of the resulting data segments were almost indistinguishable from the case where the compilation unit was just compiled once and so is not reported.

Relative repetitive processing times are shown in Figures 26, 27 and 28. The intuitive explanation of the difference between the distributions has to do with the amount of work that is eliminated from each phase. For the preprocessor, if it is possible to reuse an IR then the text of the header does not have to be read in and the preprocessed text of the header does not have to be written out. For the compiler proper, there is a reduction in the amount of input text being processed but the same output must be generated by both the conventional and the modified version of the compiler. It seems reasonable that there is a bigger improvement for the preprocessor where both the input and output text is reduced compared to the compiler for which just the size of the input text is reduced.

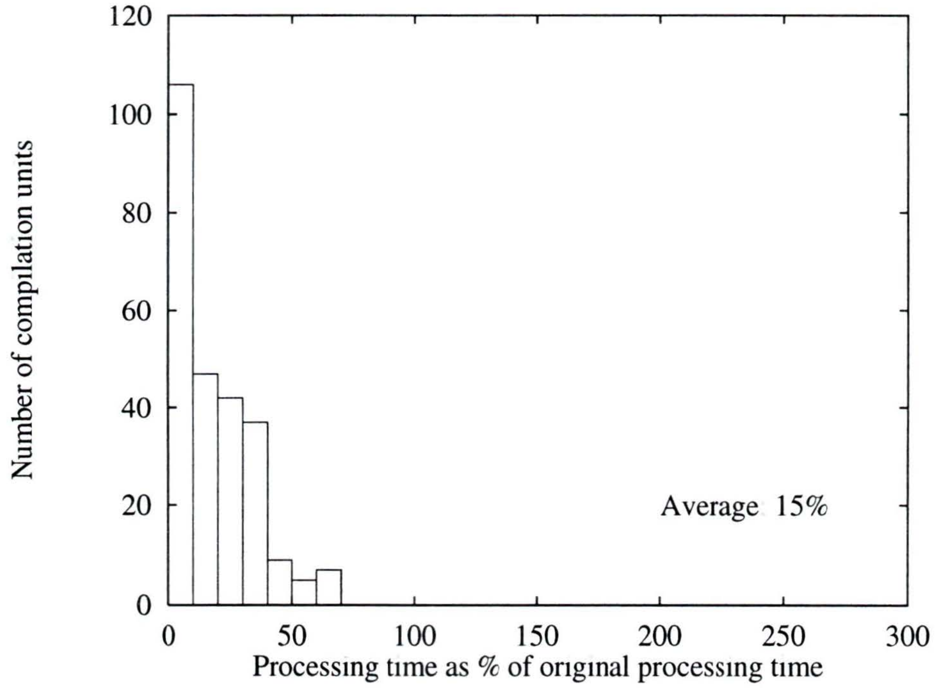


Figure 26. Distribution of relative repetitive preprocessing times

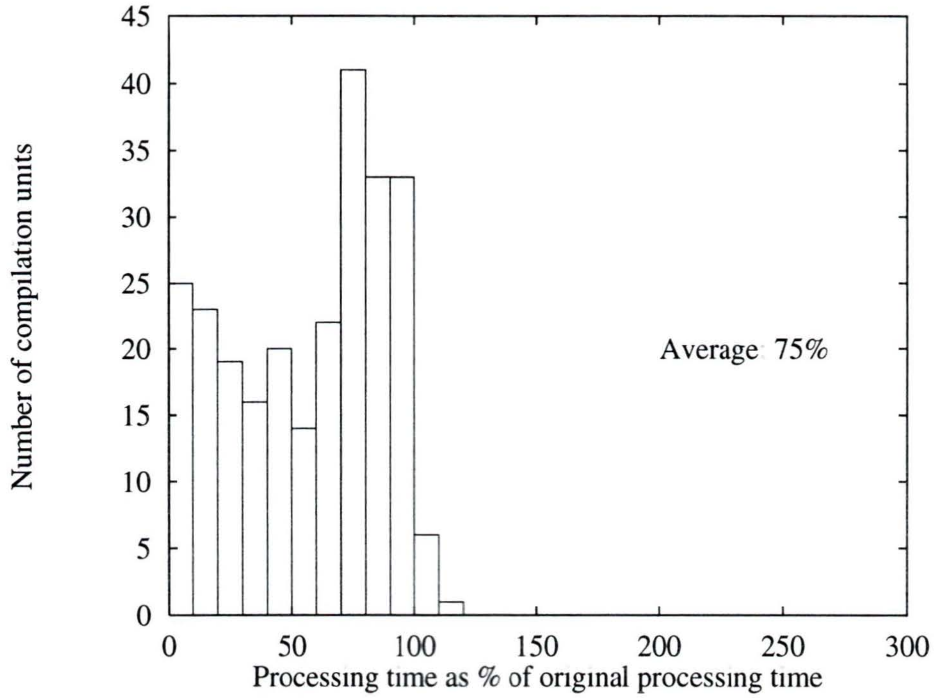


Figure 27. Distribution of relative repetitive compilation times

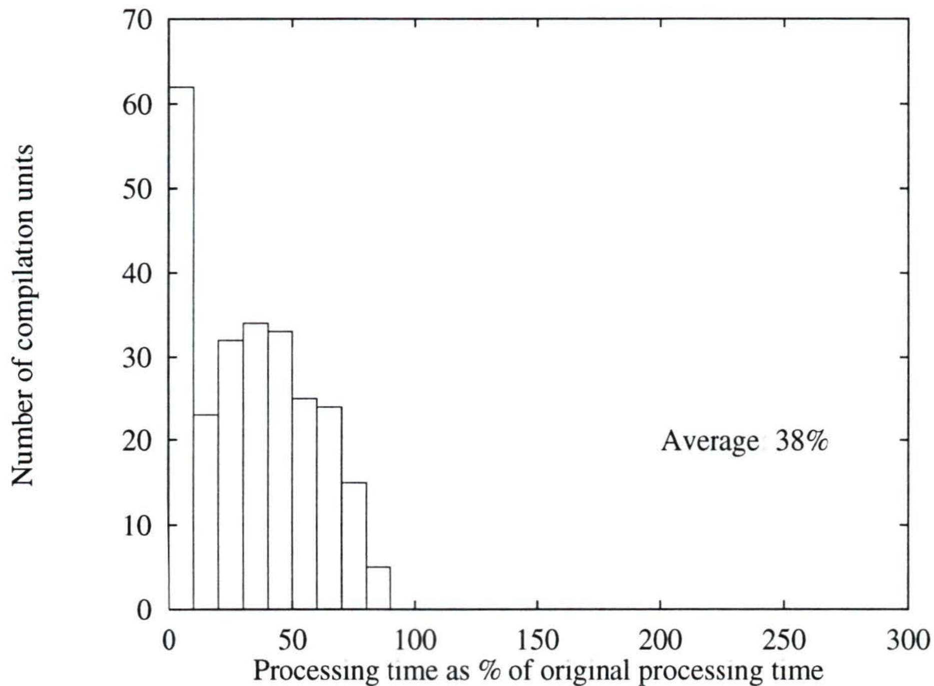


Figure 28 Distribution of relative repetitive overall times

5.3.3 Massive compilation

The measurements for this situation were made as follows for each group of related compilation units, start the compilation server, compile each compilation unit in the group recording the compilation time for each, record the final size of the data segment

The relative processing times are shown in Figures 29, 30 and 31 The preprocessing distribution is naturally slightly worse than it was for the repetitive case This is because, for each compilation unit in the massive case, there is a mix of processing the original text of the header and processing the IR of a header whereas in the repetitive case the IR of the header is always processed The performance of the compiler proper, in contrast to the repetitive case is much worse and can be attributed to three causes Firstly, the string table grows between compilations rather than being reset to empty and so the speed of string lookup gets progressively worse as the compilation server continues to run Secondly, at some arbitrary size, the string table is discarded completely along with all the IRs and so some commonly used IRs are generated several times Finally, like the preprocessor, there

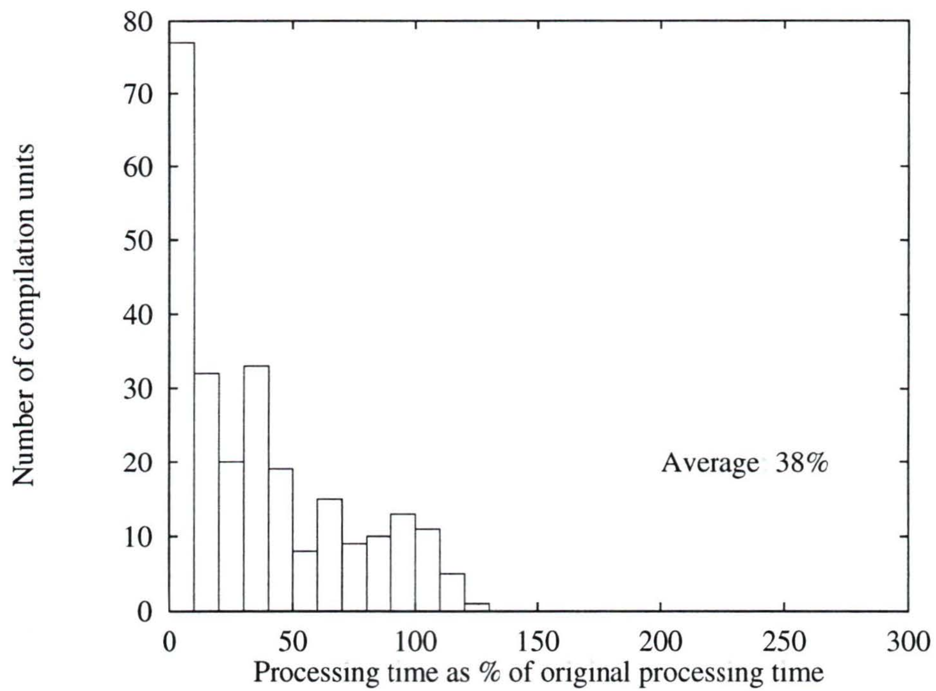


Figure 29. Distribution of relative massive preprocessing times

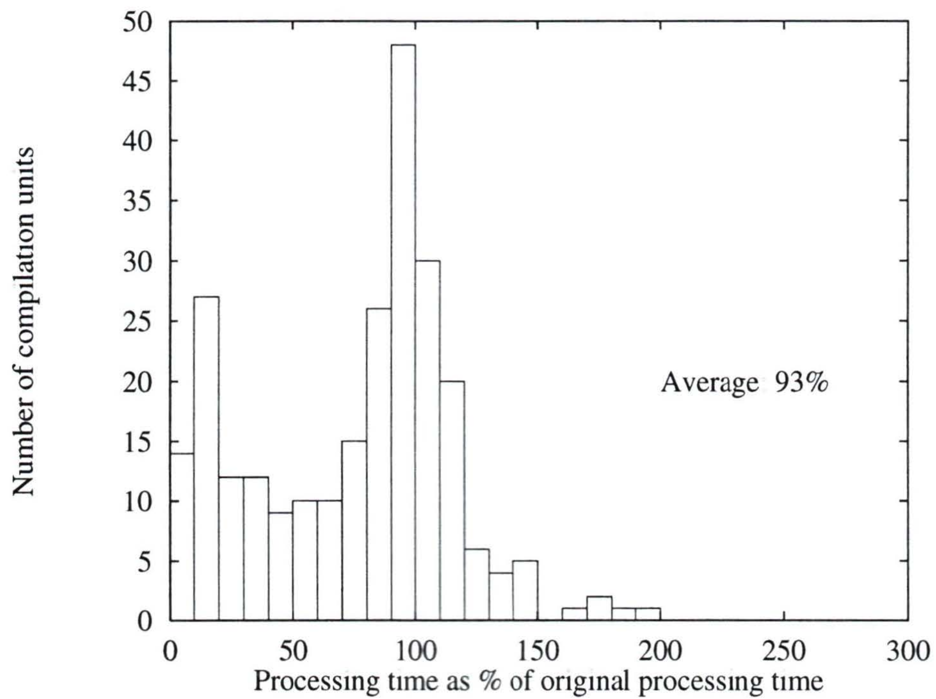


Figure 30. Distribution of relative massive compilation times

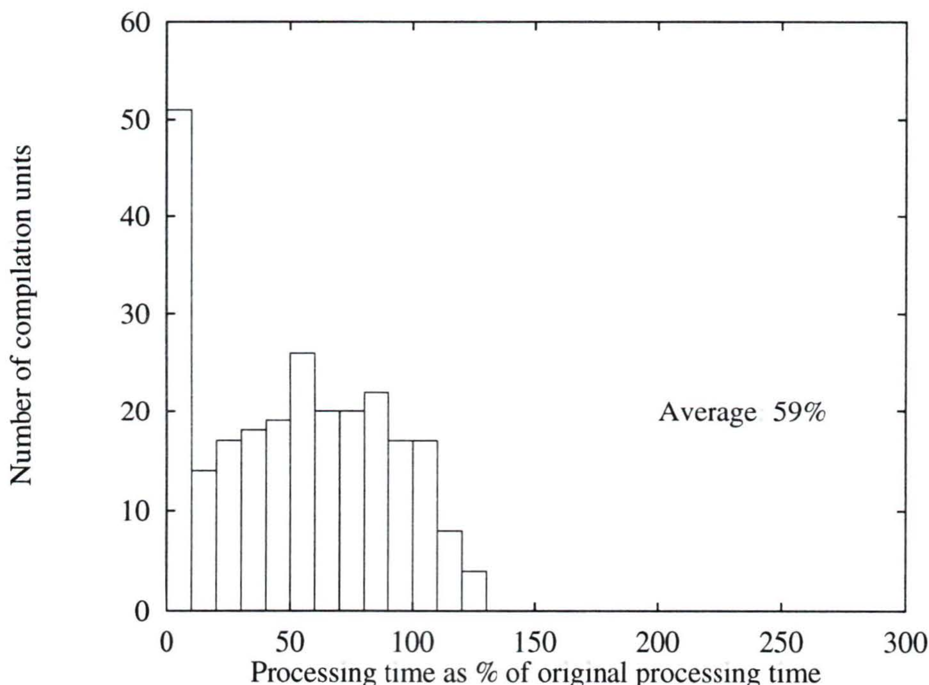


Figure 31. Distribution of relative massive overall times

is a mix of processing the original text of a header and the IR of a header, the situation is somewhat worse for the compiler proper since the context requirements for reusing and IR are more restrictive than for the preprocessor, in particular the compiler may require the reuse of several other IRs in order to reuse an IR whereas the preprocessor only demands that the state of the preprocessor be compatible with the IR to be reused regardless of what other IRs have been reused.

It is obviously inappropriate to compare the data segment size of the compilation server after compiling a large number of files to the size of the original compiler after compiling a single file. Instead we simply report that for our test cases the preprocessor server and compilation server reached maximum data segment sizes of 3 and 5 megabytes respectively.

As a final measure of the usefulness of the technique, we look at the amount of number of times IRs get reused. In Table 4 the number of IRs created and used in the various situations tested is shown. The repetitive situation generated many more IRs because each compilation unit was compiled twice and then the compilation server was restarted so any IR could be reused at most once. The number of compiler IRs is smaller than the number of preprocessor IRs because the contents of some header files are not strictly declarative, i.e. they contain function bodies, static initializers, or consist of one or more fragments of declarations. Still, the number of compiler IRs is quite good relative to the number of preprocessor IRs. The reason we did not get 100% usage of the compiler IRs in the repetitive situation was because some headers made use of identifiers declared in the main text of the compilation unit and not in another header, for example, in the following compilation unit:

Situation	Number of preprocessor IRs	Number of compiler IRs	Number of uses of compiler IRs
Repetitive compilation	1329	1226	1177
Massive compilation	494	409	657

Table 4: IR usage

```
typedef int SomeType;
#include "foo.h"
```

where the contents of "`foo.h`" is:

```
SomeType SomeVariable;
```

the IR of "`foo.h`" can never be reused because SomeType was declared in the main text of a compilation unit rather than another header file. Interestingly we don't seem to get that much leverage out of IRs in the massive situation, each IR is used on average 1.6 times. Although we have not measured it, we suspect there are a small number of IRs which get reused a lot and a large number of IRs which never get reused.

6 Conclusions and future work

The main contribution of this thesis was to demonstrate that C compilation units can be processed more quickly if the compiler's internal representation of header files is retained for reuse among compilations. A secondary result was that, with some compromises, an existing conventional compiler could be modified to achieve these results; it was not necessary to write a compiler from scratch. In terms of future work, we consider possible improvements to our current implementation and speculate on the applicability of the technique to other programming languages.

6.1 A faster C compiler

We began with a preprocessor/compiler combination which was comparable in terms of compilation speed with both the bundled and unbundled compilers available on the test system as well as the GNU C compiler. The modified preprocessor and compiler were noticeably faster on average than the original implementation. The test compilation units were compiled an average of 62 percent faster under the best possible conditions where each unit was able to reuse the maximum number of internal representations possible under the scheme. We consider this to be typical of the situation a programmer finds herself in while debugging a single unit which might be compiled over and over. Another case that was investigated was the successive compilation of several related compilation units. Under this situation, units were compiled an average of 41 percent faster. We also

considered the worst case, the situation in which no internal representation of headers were available to the compilation unit. In this case compilation was 5 percent slower on average.

The time savings do come at a price, however, and in this case the cost manifests itself in terms of memory usage. The compilation of a single unit by the compilation server required around 1.3 times more storage on average than the original compiler. The amount of storage continues to increase in the massive compilation case as the collection of internal representations of headers is built up.

The primary compromise made in this implementation was to discard all internal representations of headers periodically in the compiler proper when the string table grows beyond some threshold. Although not an ideal solution, this seems to work well enough in practice. A secondary compromise was the introduction of a lazy strategy for checking the validity of cached declarations. This yielded an implementation which no longer conforms exactly to the ANSI standard, however any inconsistencies are harmless, and the technique greatly speeds up the compilation of certain kinds of compilation units (a reduction in time proportional to the load factor of the hash table used to implement the symbol table for the compiler).

6.2 Future work

As far as this implementation is concerned, one area of interest would be finding a reasonable solution to the string deallocation problem described above. Another area of study would be examining the possibilities for retaining the internal representations for constant initializers and function bodies (see the section on C++ below). Other more general considerations include issues relating to industrial implementations such as mechanisms for scheduling multiple compilation requests and managing security between clients.

This work also only considered the time saved in generating assembly code. If the assembly code is generated successfully then that code will likely be assembled into object code.

If the assembly is successful then the object code will likely be linked to create an executable. Our technique obviously has no effect on the speed of the assembly or linking phase. In order to observe the real benefit of our technique, it would be useful to instrument a "live" compiler to count the number of times the compiler, assembler, and linker are each invoked.

An obvious question that arises is: what are the implications for compilation servers for other programming languages?

6.2.1 C++

If C++ enjoys continued success then we can expect to see systems implemented in C++ as large as or larger than those already implemented in C. Unfortunately the high degree of similarity between C and C++ will likely result in a high degree of similarity in the way C++ compilers are implemented. If C++ programs are to be compiled efficiently, a server scheme should be seriously investigated. C++ shares all of the context problems associated with C since header files are allowed to contain arbitrary text. C++ has the additional problem that "good" C++ programs will contain some member function definitions right in the class declarations so that references to these functions can be inlined. As a result, any practical compilation sever for C++ will have to retain the internal representation of function bodies as well.

6.2.2 Ada

Ignoring generics and subunits in order to simplify the discussion, the smallest constructs of Ada that can be separately compiled are subprogram declarations, package declarations, subprogram bodies, and package bodies. A package, at the outermost level of a program, is a construct for grouping related objects such as subprograms, variables, types, and other packages, the subset of these objects intended to be visible to other top level packages or subprograms constitute the interface of the package. The interface of a package consists of all objects declared in the package specification. For some packages the package specification completely describes the package, other packages require a body.

Package specifications are restricted in the sense that they cannot contain subprogram or package bodies. Package bodies contain corresponding bodies for subprograms and packages appearing in the package specification as well as declarations of other objects required privately by the package.

A sample Ada program is shown in Figure 32. The lines of dashes delimit the smallest

```
package stack is
      procedure push( x : integer );
      function pop return integer;
end;
--------------------
package body stack is
      s : array (1..100) of integer;
      top : integer range 0..100 := 0;
      procedure push( x : integer ) is
            -- ...
      end push;
      function pop return integer is
            -- ...
      end pop;
end;
--------------------
```

```
with stack;
procedure main is
      -- declarations
      elem : integer;
begin
      -- statements
      -- ...
      stack.push( 7 );
      -- ...
      elem := stack.pop;
      -- ...
end;
```

Figure 32. Ada example

possible textual fragments that could be separately compiled according to the language definition (in theory, with enough contextual information, individual symbols could be separately compiled). The first unit is the package specification and 'push' and 'pop' form the interface of the package. The second unit is the package body which additionally defines 's' and 'top' which are private to the package. The third unit is a subprogram since a complete Ada program requires at least one subprogram. The `with` clause appearing before the subprogram 'main' illustrates how the interface of a package is made available to other packages and subprograms; in this case the with clause signals the compiler to make the interface of the package 'stack' available to the subprogram 'main'. In this example the package specification and body for 'stack' depend on the same set of packages (in this case no other packages); in general, however, package bodies can depend on

a superset of the packages appearing in their corresponding specifications, likewise for subprogram specifications and bodies. Every body implicitly depends on the packages appearing in the `with` clause of its corresponding declaration so only any additional packages need to be specified.

Type checking between compilation units by an Ada compiler is facilitated by what is described as a library file in the Ada standard [21]. The standard does not state explicitly what is contained in a library file, however it implies that it contains at least symbol tables for each successfully compiled unit and information regarding the order of compilation. The idea is that when a compilation unit is compiled the library file is read, in whole or in part, for the purposes of type checking across compilation units and is updated on the successful compilation of a unit. The existence of this external file imposes an order in which units must be compiled. The standard states that consistency is guaranteed provided specifications are compiled before their corresponding bodies and a unit is compiled after all units named in its with clause. Similar rules apply to recompilations but the standard allows that a particular implementation may skip compilations if it can deduce that some units are not affected by the change (see Chapter 2). In practice the library file exists in a compressed format [1] rather than containing the original textual declarations in order to speed up compilation. At least one implementation, the GNU Ada compiler [3], does not use a library file at all and processes library information directly from the appropriate source file. This method has the disadvantage that source files have to be named in such a way that the compiler can find the correct text when processing a with clause; it also restricts files to containing exactly one compilation unit. The GNU compiler also stores source file dependencies directly in the object file so that a set of object files can be examined to see if they depend on a consistent set of sources before being linked together. This method makes some compilation order issues irrelevant.

We would expect a compilation server for a conventional Ada compiler to offer a modest improvement in compilation speed. The library file is already in an intermediate format, however we could eliminate the rereading and reprocessing of commonly used interfaces by retaining the internal representation in the compiler. A compilation server would also

be relatively easy to implement since the module interface is a source language construct and the context of an interface is made explicit by the `with` clause unlike C and C++ where the context depends on where an interface gets included.

6.2.3 Modula-2

The smallest constructs of Modula-2 that can be separately compiled are definition modules and implementation modules. A module, at the outermost level of a program, is a construct for grouping related objects such as procedures, variables, types and other modules, the subset of objects intended to be visible to other top level modules constitute the interface of the module. The interface of a module consists of all objects declared in the definition module. Every definition module has a corresponding implementation module. Definition modules are restricted in the sense that they cannot contain procedure bodies or local module declarations. Implementation modules contain corresponding bodies for procedures appearing in the definition module as well as declarations of other objects required privately by the module.

A sample Modula-2 program is shown Figure 33. The lines of asterisks delimit the small-

```
DEFINITION MODULE stack;
    PROCEDURE push( x: INTEGER );
    PROCEDURE pop() : INTEGER;
END stack.
(********************)
IMPLEMENTATION MODULE stack;
    VAR s: ARRAY [1..100] OF INTEGER;
    VAR top: [0..100];
    PROCEDURE push( x: INTEGER );
        (* ... *)
    END push;
    PROCEDURE pop() : INTEGER;
        (* ... *)
    END pop;
BEGIN
    top := 0;
END stack.
(********************)
```

```
MODULE main;
IMPORT stack;
    (* declarations *)
    VAR elem: INTEGER;
BEGIN
    (* statements *)
    (* ... *)
    stack.push( 7 );
    (* ... *)
    elem := stack.pop();
    (* ... *)
END main.
```

Figure 33. Modula-2 example

est possible textual fragments that could be separately compiled according to the language definition The first unit is the definition module for 'stack', the second unit is the implementation module for 'stack', and the third unit is a main module which performs a similar function to the subprogram 'main' in the Ada example The `IMPORT` clause appearing in the main module illustrates how modules access the interfaces of other modules In this case it is expected that the declarations or statements of the main module will directly reference objects in the interface of the 'stack' module As in Ada, an implementation module can depend on a superset of modules upon which its corresponding definition module depends Also, implementation modules implicitly depend on the modules specified in their corresponding definition modules and need only explicitly mention any additional modules

Type checking between compilation units by Modula-2 compilers is typically achieved through the use of symbol files A symbol file is created for each definition module compiled and contains an intermediate representation of the corresponding text The existence of these intermediate files imposes a compilation order on compilation units similar to that for Ada Implementations use a broad range of symbol file formats At one extreme there is at least one implementation [5] that does not use symbol files at all and processes import clauses by recompiling the text directly from the file containing the module definition At the other end of the spectrum, another implementation [9] uses a symbol file which is close in format to the internal representation of the symbol table used by the compiler and is complete in the sense that it contains all the symbols from all definition modules appearing in the transitive closure of its text's import clauses

The benefits of a compilation server for Modula-2 are quite similar to Ada essentially avoiding rereading and reprocessing commonly used symbol files Again we expect implementation to be relative easy since the module interface is an explicit source language construct and context is made explicit by the `IMPORT` clause

Ultimately, we have demonstrated that retaining information between invocations of a compiler can significantly speed up compilations in most practical cases As a result we

believe that this should be seriously kept in mind when considering the design or redesign of new or existing compilers

References

1 R Adams, W Tichy and A Weinert, 'The cost of selective recompilation and environment processing', *ACM Transactions on Software Engineering and Methodology,* 3, (1), 3-28 (1994)

2 Borland International, Inc , *Borland C++ User's Guide*, Scotts Valley, CA, 1993

3 C Comar, F Gasperoni and E Schonberg, *The GNAT project A GNU-Ada9X compiler,* unpublished, 1993

4 S I Feldman, 'Make - a program for maintaining computer programs', *Software-Practice and Experience,* 9, (3), 255-265 (1979)

5 D G Foster, 'Separate compilation in a Modula-2 compiler', *Software-Practice and Experience,* 16, (2), 101-106 (1986)

6 A Fyfe, I Soleimanipour and V Tatkar, 'Compiling from Saved State Fast Incremental Compilation with Traditional UNIX® Compilers', *Proceedings of the Winter 1991 USENIX Conference*, pp 161-171

7 C W Fraser and D R Hanson, *A Retargetable C Compiler Design and Implementation*, Benjamin/Cummings, in press 1994

8. C. W. Fraser and D. R. Hanson, *A Retargetable Compiler for ANSI C*, Research Report CS-TR-303-91, Department of Computer Science Princeton University, 1991.

9. J. Gutknecht, 'Separate compilation in Modula-2: an approach to efficient symbol files', *IEEE Software,* 3, (11), 29-38 (1986).

10. D. R. Hanson, 'Fast allocation and deallocation of memory based on object lifetimes', *Software-Practice and Experience*, 20, (1), 5-12 (1990).

11. R. Hood, K. Kennedy and H. A. Müller, 'Efficient Recompilation of Module Interfaces in a Software Development Environment', *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 180-189.

12. R. F. Kamel, 'Effect of modularity on system evolution', *IEEE Software*, 4, (1), 48-54. (1987).

13. Lattice, Inc., *Lattice C Compiler User's Manual*, Lombard, IL, 1988.

14. A. Litman, 'An implementation of precompiled headers', *Software-Practice and Experience,* 23, (5), 341-350 (1993).

15. P. J. Moylan, *The Case Against C,* Technical Report EE9240, Department of Electrical and Computer Engineering, University of Newcastle, 1992.

16. T. Onodera, 'Reducing compilation time by a compilation server', *Software-Practice and Experience,* 23, (3), 477-485 (1993).

17. R. W. Schwanke and G. E. Kaiser, 'Smarter recompilation', *ACM Transactions on Programming Languages and Systems,* 10, (4), 627-632 (1988).

18. B. Stroustrup, *The C++ Programming Language,* Second Edition, Addison-Wes-

ley, Reading, MA, 1991

19 Symantec Corporation, *THINK C User's Guide*, Cupertino, CA, 1994

20 W F Tichy, 'Smart recompilation', *ACM Transactions on Programming Languages and Systems,* 8, (3) 273-291 (1986)

21 U S Department of Defence, *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*, Springer-Verlag, New York, 1983

22 N Wirth, *Programming in Modula-2,* Third, Corrected Edition, Springer-Verlag, New York, 1983

23 XJ311 Committee, *Draft Proposed American National Standard for Information Systems -- Programming Language C*, 1988

VITA

Surname: Koehler   Given Names: Brian Keith

Place of Birth: Toronto

Educational Institutions Attended:

University of Victoria   1993 to 1994

University of Waterloo   1987 to 1992

Degrees Awarded:

B.Math. (Honours) University of Waterloo   1992

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: A Caching Compiler for C

Author

(Signature)

BRIAN KEITH KOEHLER

(Name in Block Letters)

DECEMBER 14, 1994

(Date)