SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(3), 341–350 (MARCH 1993)

An Implementation of Precompiled Headers

ANDY LITMAN*
NeXT Computer, Redwood City, CA 94063, U.S.A.

SUMMARY

Compilation time can be improved by precompiling interfaces so that the compiler can avoid processing unreferenced declarations. However, in C-based languages precompiling a header is difficult because a header can have several meanings depending on the context in which it is included. We present an *ad hoc* solution to this problem, and give the results of our precompilation strategy, which improved compile times by 25 to 65 per cent over compilation without precompiled headers.

KEY WORDS: Precompiled header Compilation time Preprocessor Conditional compilation

INTRODUCTION

In many development environments for traditional languages, a large percentage of compilation time is spent processing declarations in interfaces. Since in general interfaces change infrequently compared to implementations, compile time can be improved by keeping interfaces in a precompiled or preprocessed form.

However, there are a number of problems with this approach. An obvious problem is maintaining consistency between interfaces and their precompiled forms; straightforward solutions are possible based on modification times.

More subtle problems occur when the language allows interfaces to be context dependent, as conditional compilation does in C-based languages. If the meaning of the interface can vary depending on the context in which it is used, then the precompiled interface must store all possible meanings, or the compiler must 'context check' the precompiled interface with respect to the current context. Several commercial compilation systems take the latter approach.[1] In these systems, checking is simplified by allowing a compilation unit to include only one precompiled interface, and allowing it only at the beginning of the compilation unit, so that the context is empty or nearly empty. It may be the case that without this restriction, checking would take more compilation time than is saved by using precompiled interfaces. Our system uses more sophisticated but still fast checking, and is able to remove these restrictions on usage.

There are many trade-offs to be made in the format of a precompiled interface, which is essentially a symbol table. Size is an important consideration; precompiled interfaces for system libraries can consume several megabytes, effectively restricting

* Current Address: Computer Science Department FR-35, University of Washington, Seattle, WA 98195, U.S.A.

0038–0644/93/030341–10$10.00

Received 2 February 1992
Revised 19 October 1992

their number. Some of our design decisions were based on our desire to access precompiled interfaces from a preprocessor, without making any modifications to the compiler itself. For instance, our format had to allow declarations to be reconstituted as they appeared in the original interface. The format also had to allow quick name look-up based on the scoping rules of the language. This was complicated by the fact that our target language was polymorphic, so look-up of a method name might yield several declarations.

The C preprocessor plays a special role in our compilation process. Assume that some interfaces have been precompiled so that look-up of declarations for particular identifiers is easy. When the source file is preprocessed, the only declarations from precompiled interfaces that appear in the output are those necessary for its correct compilation. The output of the preprocessor is compiled as usual.

Precompiled interfaces have made several improvements to our development environment: they have

(a) reduced overall compile time for projects by 25 to 65 per cent
(b) reduced symbol table sizes by 40 per cent and, indirectly, reduced link times by 10 per cent
(c) enabled programmers to use large interfaces with acceptable compile times
(d) given fast recognition of syntax errors during compilation
(e) provided a foundation for other language-based tools
(f) provided an incentive for 'cleaning up' interfaces.

In this paper, we first describe our development environment and our goals in designing precompiled interfaces. Then we describe our implementation and our solutions to the problems introduced above. Finally we show our results and discuss follow-up work.

BACKGROUND

The NeXT development environment provides a rich set of tools for developers. The language tools include compilers for Objective-C and C++, a debugger, and several browsers for both static and dynamic program examination. Objective-C,[2] a minimal object-oriented extension to ANSI C,[3] is the language most of our developers use, although C++[4] is becoming more popular. The speed of the Objective-C compiler is generally considered quite good, but compile time is still considered a bottleneck in the development cycle.

Therefore, the primary goal for precompiled interfaces (hereafter called precompiled headers) was to reduce compile time for Objective-C and ANSI C, and eventually for C++ as well. Secondarily, usage of precompiled headers should be transparent to the programmer. The compiler decides if a suitable precompiled header is available, and if so uses it without informing the programmer. We were willing to sacrifice some transparency for speed, but that turned out not to be necessary.

Probably the earliest system to use precompiled interfaces was Mesa.[5,6] Interfaces were separately compiled into symbol tables, which could then be used in compiling implementation modules. The compiler type-checked any references to interface items from the implementation, although each interface itself had to be type correct when it was precompiled.

A key difference between Mesa and C is that Mesa interfaces are context

independent, which makes it feasible to compile them uniquely into symbol tables. In contrast, C headers are context sensitive and are usually included textually. One reason textual inclusion does not scale well is that mature systems tend to have large numbers of long interfaces. At NeXT it is not uncommon for a module to include headers with 10 times its number of lines. It is possible for programmers to minimize the number and size of headers, but without tools this chore is tedious. When textual inclusion is used a significant fraction of compilation time is spent on the parsing and semantic analysis of headers.

Object-oriented languages, such as Objective-C and C++, complicate precompilation schemes such as ours in two ways. First, a class groups declarations so that a reference to some part of a class, such as a method, may require that the entire class be declared. Secondly, dynamic binding and polymorphism may cause the compiler to demand more declarations than are actually used by the program. For example, the Objective-C compiler warns of a message name duplicated in two classes if that message is sent to an untyped object; both classes with the duplicate must be declared to the compiler. These object-oriented features tend to increase the size of the preprocessor output and slow overall compilation time.

Precompiled headers can be used in conjunction with other consistent compilation tools. There is evidence that most recompilations are unnecessary;[7] time-stamps[8,9] and dataflow techniques,[10,11] for example, have been used to avoid unnecessary recompilation. Our goal was merely to speed up, not to avoid, compilation.

Incremental compilation[12,13] and dynamic compilation[14] have been used to improve compile times, but these approaches were inappropriate for our environment. Both would have required major changes to the compiler and substantial environment support, and neither handles preprocessing as well as we would like. Furthermore, dynamic compilation has an unacceptable run-time performance penalty.

IMPLEMENTATION

Our initial approach to precompiled headers was to modify the compiler to dump a memory image of its internal data structures as it compiled a header. Unfortunately, these dumps were so large that it would have been impractical to keep more than a few on a typical machine. Furthermore, this format contained none of the information necessary for context checking. We turned to a different approach that we hypothesized would be more space efficient and, in the long run, might provide a basis for other language tools.

We call our approach *smart preprocessing*. In this approach the standard C preprocessor is replaced with a preprocessor that does the following:

1. Preprocess as usual, noting precompiled headers that are included.
2. Parse the source file and any non-precompiled headers. In this stage the source is lexically analyzed and parsed as it would be in the front end of the compiler, and a table of declarations is built. Parsing is necessary to detect undeclared identifiers.
3. During parsing, look up identifiers that might be undeclared in the table of declarations. If the identifier is not in the table, look up the declaration in any precompiled headers seen so far. The identifier is added to the declarations

table, the precompiled declaration is marked for output, and the declaration is itself parsed to detect further undeclared identifiers.

4. Iterate step 3 until the parse is completed and there are no undeclared identifiers.

Here is a file before and after smart preprocessing, assuming stdio.h has been precompiled (the number 84 is a line number within stdio.h):

```
#import ⟨stdio.h⟩

main() {
  printf("hello world");
}
```

```
#1 "hello.c"
#84 "/usr/include/stdio.h"
extern int printf(const char *format, ...);
#1 "hello.c"

main() {
  printf("hello world");
}
```

The key objective in smart preprocessing is extracting from precompiled headers only those declarations that will actually be needed by the compiler. The logic is implemented by three functions called by various parser actions:

```
lookup_declaration (identifier)
  If identifier names a class
    recursively look up its superclass
    parse instance variable declarations but not method declarations
  Otherwise
    recursively parse declaration

lookup_tag (identifier)                    %struct, union, or enum tag
  recursively parse declaration

lookup_method (identifier)
  For each method named identifier         %manage polymorphism
    recursively look up its class
    recursively parse method declaration
```

For example, in the 'hello world' program above, lookup_declaration("printf") would be called from the parser, printf would be extracted from the precompiled header, and the declaration of printf would be recursively parsed.

Notice that looking up a class does not automatically look up all of the methods for the class, since the compiler does not necessarily need them. For a message send, however, all methods with the given message name are looked up regardless of class. This is because the smart preprocessor does not do type analysis, so even though the receiver may be statically typed, the preprocessor does not know which class actually implements a given message. Even if the preprocessor did type analysis, it would have to remain conservative for untyped objects.

Some of the recursive parsing in the look-up routines above could be avoided by storing with each declaration in a precompiled header a list of the identifiers it depends upon. This approach would save a small amount of parsing time at the cost of some space in the precompiled header.

Once the source has been entirely parsed and all references satisfied, the preprocessed source is sent to the output stream. Declarations from the precompiled header are output in the order they appeared in the original header, with correct file and line locations for each token.

The implementation of smart preprocessing required a substantial programming effort. Most of this effort went into a development toolkit consisting of a lexical analyzer, a recursive descent parser, and a preprocessor. The lexical analyzer produces a stream of tokens, the parser consumes one, and the preprocessor is both a consumer and producer. By hooking producers and consumers together, smart preprocessing is done in a single pass over the source, followed by an output phase.

From our experience, smart preprocessing has little effect on debugging. During debugging, programmers rarely want access to declarations that were not referenced directly or indirectly anywhere in the source file. However, there are plans to enhance the debugger to read precompiled headers so that it can provide information about all the declarations in our environment. The one case where smart preprocessing does have a noticeable effect on debugging is when an object is declared to be of a particular class but the object is actually an instance of a subclass. In this case, the symbols for the subclass may not be available. As yet we have no solution to this problem that does not require type analysis.

BUILDING PRECOMPILED HEADERS

The preprocessor can precompile any header that parses correctly. Note that this restriction is significant for C headers, since undefined types cause syntax errors. The smart preprocessor can safely assume that declarations extracted from a precompiled header parse correctly. There are no restrictions on macro definitions in precompiled headers as long as macro expanded declarations parse correctly.

A header is *context dependent* if the declarations in the header may change depending on the context in which it is included. Most uses of conditional compilation and macro expansions cause context dependence. For instance, the following header is context dependent:

```
#ifdef DEBUG
int a;
#else
int b;
#endif
```

The context at any point is determined by the macros that are defined there. A precompiled header must be created in a context compatible with that in which it is used. By passing switches the preprocessor, any set of macros can be predefined, creating a context in which the precompiled header is built.

The file format of precompiled headers consists of:

(a) preprocessed tokens for the header, with source back-mappings

(b) a string table
(c) macro definitions at the entry and exit of the header
(d) external declarations of typedefs, classes, enumeration constants, etc.
(e) tag declarations for structs, unions, and enums
(f) method declarations
(g) nested header names, time-stamps, and nesting level.

Declarations are sorted alphabetically, whereas nested headers are sorted in order of inclusion. During smart preprocessing, a precompiled header is read-only, and relocation is done on the fly as fields are accessed.

The size of a precompiled header is consistently less than twice the size of the text of the preprocessed header. Our largest header expands to a quarter megabyte of text, and the precompiled form is less than half a megabyte. This size makes it feasible to ship precompiled interfaces for all our major libraries; it is neither feasible nor desirable to ship precompiled forms of all headers.

CONTEXT CHECKING

When the smart preprocessor encounters an include directive, it first looks for a precompiled version of the header. If one is found, it checks whether the context is equivalent to the context in which the precompiled header was built. If any of the following problems occur, the non-precompiled form is included:

1. A header that was included by the precompiled header could not be found in the file-system to verify its modification time, or the modification time did not match. In practice, this problem never occurs for precompiled headers that are part of the release, and rarely occurs when programmers build their own precompiled headers.
2. A macro was defined when the precompiled header was built, but is not defined in the current context. This is a problem only if the macro was actually referenced somewhere in the precompiled header; each macro in the precompiled header carries a referenced flag to enable this check. Although our headers define many macros, fewer than 10 per cent of those that are defined in a particular header are actually used in that or any other header. This problem also occurs rarely.
3. A macro was undefined when the precompiled header was built, but is defined in the current context. This is a problem only if there might have been an invocation of the macro in the precompiled header. Such an invocation is discovered by scanning the string table of the precompiled header, which contains every identifier in the header. This check is conservative because it is not certain that the macro would actually have been invoked; however, our usage patterns indicate that the check is not overly conservative. This check sometimes fails when a precompiled header is included after other headers in the source file, but a satisfactory work-around for the programmer is to include precompiled headers in the source file before any other headers, as the Borland approach requires. This is a reasonably efficient solution to the context dependence problem; in practice the check is fast and safe. However, in the future we plan to experiment with more sophisticated solutions, such as keeping separate string tables for each nested header.

USAGE

If precompiled headers are big (contain many headers, that is), a given implementation file may include fewer precompiled headers, and generally compiles faster. This is because the look-up algorithms for precompiled headers are logarithmic with respect to the number of declarations in the header. However, bigger precompiled headers make name conflicts more likely. Name conflicts manifest themselves as preprocessing errors, syntax errors, or semantic errors.

For example, if you were to combine all the headers for a project, including system headers (headers included with ⟨⟩ delimiters), into a single precompiled header, it is likely that there would be a name conflict. There may be a macro defined that happens to match one of your local identifiers, or there may be a public struct declared that happens to match one of your private struct names. The conflicts may be resolved by renaming identifiers, or removing a conflicting header from the precompiled header. This problem exists regardless of whether you precompile or not, but because precompiled headers allow larger headers to be included efficiently, conflicts tend to occur more often.

Another disadvantage to big precompiled headers is that the resulting file dependencies may lead to unnecessary recompilation. If all implementation files in a project depend on a single precompiled header that in turn depends on all headers in the project, then changing a header requires recompilation of the entire project. A better approach is to build a precompiled header containing all system headers used by a project, and perhaps also a separate precompiled header for the local headers in the project. We recommend that during development, while local headers are unstable, precompiled headers be used only for system files. When local headers have stabilized, they may be precompiled.

For small projects, casual users need not bother building their own precompiled headers. They can just use the precompiled headers that are bundled with the release. A dozen or so large-grain precompiled headers are built as part of the software release process, and in fact these are transparently used in building the remainder of the system.

RESULTS

We have converted several projects to use precompiled headers, and improvements in overall compile time for the projects have ranged from 25 to 65 per cent (see Table I). There are few files whose compile time did not improve. In those cases, files that included a small number of headers were modified to include a single precompiled header that combined all of the headers for the project. This shows that there is sometimes a break-even point when modifying projects to use precompiled headers, based on the size of the project headers, that can only be determined by benchmarking each project.

When we were designing precompiled headers, we suspected that removing declarations during smart preprocessing would affect symbol table sizes. The effect proved to be dramatic. If each of the headers in our largest library, appkit, was separately precompiled, the object files in appkit had 40 per cent fewer symbols. This difference had the side-effect of reducing link time for appkit by 10 per cent, since the linker spends a good deal of its time copying symbols.

Another benchmark of symbol table sizes was less encouraging. When the Editor,

Table I. Compile times. This table gives compile times for several small applications, although the results are generally similar both for larger projects and for individual files. The columns are: program name, the number of lines in the implementation files excluding headers, the preprocessing time and total compilation time using the old compilation system, the preprocessing time and total compilation time using smart compilation, the ratio of smart preprocessing time to old preprocessing time, and the ratio of smart compilation time to old compilation time. Smart preprocessing is generally faster than the old preprocessing, although it can be slower as one case shows. The total compile time is uniformly faster for smart compilation

Program	Lines	Old Cpp (s)	Old comp (s)	Smart Cpp (s)	Smart comp (s)	Cpp ratio	Comp ratio
Acceptor	1091	8·0	36·2	5·6	20·1	0·70	0·55
BusyBox	2446	6·8	34·1	5·9	22·6	0·87	0·66
CalculatorLab	285	1·7	8·1	1·1	4·3	0·65	0·53
CompositeLab	365	1·2	7·3	1·0	4·5	0·83	0·62
Draw	9134	22·2	136·1	21·4	104·5	0·96	0·77
Graph	3020	4·7	26·5	4·1	19·2	0·87	0·72
Lines	618	3·8	16·0	1·7	6·7	0·45	0·42
Mandelbrot	1400	7·2	31·0	3·4	14·9	0·47	0·48
MidiDriver	1231	2·9	10·5	1·4	6·1	0·48	0·58
PaintLab	417	1·9	10·7	1·3	5·7	0·68	0·53
ScreenSender	1082	7·1	32·9	3·3	13·3	0·46	0·40
ScrollDoodScroll	1054	5·1	27·7	4·8	18·2	0·94	0·66
SortingInAction	2244	8·4	36·1	6·2	23·6	0·74	0·65
SoundEditor	592	4·6	18·8	2·0	9·1	0·43	0·48
Subprocess	570	3·9	16·0	2·1	8·1	0·54	0·51
TextLab	319	5·6	23·7	2·3	8·5	0·41	0·36
ToolInspector	3702	8·1	44·8	8·0	29·8	0·99	0·66
VisibleView	1132	4·9	19·6	3·6	12·8	0·73	0·65
WhatADrag	3488	6·0	35·0	6·4	24·5	1·07	0·70
Yap	1152	6·5	27·9	3·8	14·9	0·58	0·53

a relatively small application, was converted to use the precompiled appkit.h, which contains all of the headers in the appkit, symbol tables grew by about 10 per cent, although compile time was still 40 per cent faster. This effect was due to polymorphism; looking up common methods such as new iteratively looked up almost every class in appkit.h. The work-around for this problem is not trivial, since the compiler could conceivably need all of the method definitions that were found, for error diagnostics. One proposed solution is to add another criterion for looking up classes other than simply having had one of their methods referenced. Once this problem is solved, we expect to achieve symbol table savings nearly as good as we initially saw.

Another result we expect to see over time is programmers losing their fear of large interfaces. Currently, programmers have a good idea of which interfaces are ‘big’, because compile times become unbearably long when they are used. We feel that programmers will be able to write their programs faster if they can use any interfaces that are available, although a few programmers feel that time spent minimizing the number and size of interfaces they use is worth while. In the long run, our language tools should efficiently support any programming or software engineering style.

During the design phase, there was some worry over the smart preprocessor parsing the source file, because it meant syntax errors would be reported in a separate pass from semantic errors, which were left for the compiler. We felt programmers who were accustomed to a 'one pass' C compiler might be confused. This turned out not to be a bug, but a feature. Syntax errors are now reported much faster than before; in particular, the user sees the first syntax error almost immediately. Also, the Objective-C compiler generates so many spurious semantic errors from a single syntactic error that finding syntax errors first reduces the total number of error messages.

Precompiled headers provide a reasonable basis for other fast language tools. Even before smart preprocessing entirely worked a snappy browser had been built for precompiled headers, and it was apparent from the first version of this browser that it was more useful to programmers than the existing librarian tool. The reason was clear: the librarian could create an index from headers, but it was not language sensitive, and so could not distinguish between declarations and uses. Programmers found it frustrating to wade through several library entries before finding the declaration. The precompiled header browser has been integrated with the editor to provide language-sensitive search and insertion of declarations with single keystrokes.

Since context dependence is the reason for context checking, we have begun encouraging developers to remove conditional compilation and macros from their headers. This clean-up may someday allow us to make significant optimizations of context checking for internal development. Since developers probably cannot do without conditional compilation altogether, a benign alternative we have considered is allowing it only at the file level.

FOLLOW-UP

We anticipate only one major change to smart preprocessing. Currently, the source file is parsed twice, once by the smart preprocessor and again by the compiler. The smart preprocessor should be integrated with the compiler so that only a single parse is done. The major work here is mapping the compiler's LALR parser actions into our recursive descent actions.

We are considering expanding the file format of precompiled headers to store multiple streams of preprocessed tokens for multiple definitions of a particular macro. This would be useful to our operating systems group, who often do conditional compilation with a macro that describes the architecture. Clearly most of the preprocessed tokens in the streams could be shared, so this approach has a space advantage over creating multiple precompiled headers.

Using the programmatic interface to precompiled headers, we were able to quickly write a prototype program that detects API changes by comparing two precompiled headers. This is important to our documentation writers, who currently detect API changes using a combination of *ad hoc* parsing and text comparison. Our method is more reliable, and can selectively ignore declaration ordering, comments, and white space.

In conclusion, our initial effort to simply improve compile time has led to several other improvements in total turnaround time. We are able to browse library interfaces conveniently, write code faster, detect syntax errors quickly, as well as compile and link faster.

ACKNOWLEDGEMENTS

Steve Naroff implemented the recursive descent parser and header browser. Dave Moore and Matt Self contributed suggestions during the implementation. David Notkin and Kevin Sullivan gave useful comments on the paper.

REFERENCES

1. *Borland C++ User's Guide*, 'Appendix A: Precompiled Headers'.
2. NeXT Technical Documentation, *Object-Oriented Programming and Objective-C*, NeXT Computer, 1991.
3. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.
4. B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, MA, 1991.
5. C. M. Geschke, J. H. Morris and E. H. Satterthwaite, 'Early experience with Mesa', *Communications of the ACM*, **20**, (8), 540–553 (1977).
6. Richard Sweet, 'The MESA programming environment', *SIGPLAN 85 Symposium on Language Issues in Programming Environments*, 1985, pp. 216–229.
7. E. Borison, 'Program changes and the cost of selective recompilation', *Technical Report CMU-CS-89-205*, Carnegie Mellon University, 1989.
8. S. I. Feldman, 'Make—a program for maintaining computer programs', *Software Practice and Experience*, **9**, (4), 255–265 (1979).
9. W. Tichy, 'Smart recompilation', *ACM Transactions on Programming Languages and Systems*, **8**, (3), 273–291 (1986).
10. Keith D. Cooper, Ken Kennedy and Linda Torczon, 'Interprocedural optimization: eliminating unnecessary recompilation', *SIGPLAN '86 Symposium on Compiler Construction*, 1986, pp. 58–67.
11. Robert Hood, Ken Kennedy and Hausi Muller, '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.
2. G. Ross, 'Integral C—a practical environment for C programming', *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 42–48.
13. Mayer D. Schwartz, Norman M. Delisle and Vimal S. Begwani, 'Incremental compilation in Magpie', *SIGPLAN '84 Symposium on Compiler Construction*, 1984, pp. 122–131.
14. L. Peter Deutsch and Alan Schiffman, 'Efficient implementation of the Smalltalk-80 system', *Proceedings of the 11th Symposium on Principles of Programming Languages*, 1984.