Automatic Precompiled Headers: Speeding up C++ Application Build Times

Tara Krishnaswamy
Internet & IA-64 Foundations Lab, HP
tara@cup.hp.com

Abstract

This paper describes the crucial design and implementation issues that arise in building a fully automatic precompiled header mechanism for compiling industrial-strength C and C++ applications. The key challenges include designing the Makefile-transparent automation, determining the precompile-able region, capturing the compile environment and verifying it and addressing the correctness issues involved in using precompiled headers. The ensuing discussion treats the internals of the actual dumping and loading of precompiled headers as a black-box beyond a brief high-level description. An automatic precompiled header mechanism has been implemented in aCC, the HP ANSI C++ compiler, and the results of compiling real applications show that it achieves significant speedup in compile-times of real applications.

Introduction

C++ compilers are increasingly called upon to translate large applications that implement formidable computing tasks. These applications range from enterprise level software for supply-chain planning and database management to technical computing systems for CAD, mechanical design and automation and internet software for web browsing. These large software systems are structured as various C++ programs and distributed between many directories in the file-system. Each set of programs that constitutes these applications is compiled and linked into shared or archive libraries.

Also, such large systems often contain a mix of existing C and C++ code with incremental additions of fresh code for each new release of the product. As the size of these applications increases with each release, the time required to build them also increases, thereby impacting the productivity of the developers. In order to minimize this impact, many industrial systems initiate complete builds at night and require such builds to finish overnight, in spite of the applications' already large and steadily increasing size. These end-to-end builds include both the compile and the link phases, but are dominated by the compile phase.

At the source level, these large programs are modularized in traditional fashion as source-files and header-files. Source files primarily contain code that implements the behavior of the application, while header files mostly contain declarations that describe various complex types and data-structures used in the application. Such applications use both system headers that advertise the underlying operating-system API and application-specific headers that describe user-defined types and data-structures.

These header files are then programmatically internalized into one or more source files, exposing the types and data-structure interfaces to the source-files that need them. This header-file inclusion mechanism also allows the compiler to perform any static type checking mandated by the C/C++ languages. This paradigm of modularizing programs attempts to separate the interface to various data-structures and types in the application from the code that uses and manipulates it.

However, neither the C nor C++ language definitions enforce this paradigm strictly, allowing practically any legal C/C++ construct to reside in header-files. This complicates the structure of the application since C header files often contain not only type declarations but also macro definitions. Macro definitions associate symbol names, possibly with arguments, with replacement text which can contain conditional statements, loops, function calls and other executable code. C++ header files add further complexity since they contain both the interfaces to various classes and the definitions of the inline member functions of those classes.

In essence, the C++ promise of software reuse with libraries is delivered to a large extent via static header files that contain a lot of code that needs to be recompiled each time the importing translation unit is compiled. This proliferation of header files with complex code demands huge compiler effort to digest and thereby negatively impacts the compile-times of the applications. Table 1 contrasts the volume of code imported by a C++ *"Hello World!"* program against one written in C. Table 2 compares the volume of code between source files and header files for some applications. The table shows that on average about 87% of the application's code resides in header files.

From Table 2 it is apparent that in order to achieve fast and efficient compiles, a compiler must seek to minimize header-processing time. Note that although about 87% of lines in the applications stem from header files, it does not imply that an equivalent portion of compile-time will be dedicated to header processing. This is partly because header files predominantly contain declarations that don't trigger machine code generation, although debug-mode compiles may cause debug information to be emitted for these declarations. Table 3 reports the compile-time division between actual source line processing and header processing.

Note that the processing times for the header files serve only as a lower bound although they represent actual compile-times due to the fact that these numbers were gathered only for headers continuously *#include*-ed at the head of the source files. Since C and C++ allow header files to be *#include*-ed anywhere in the source file, an arbitrary sequence of header files culled by separating the interrupting source lines, may not compile successfully due to dependencies on the lexically preceding source. This is addressed in greater detail in the next section.

Table 3 clearly demonstrates that a large part of the compile times of these applications is spent in processing the *#include* header files. This argues for a compilation technique that minimizes the overhead of processing headers.

Table 1

APPLICATION	PREPROCESSED LINES P_l = wc -l *.i	SOURCE LINES S_l = wc -l *.C	HEADER LINES $H_l = P_l - S_l$
Hw.c	341	6	335
Hw.C	679	6	673

Table 2

APPLICATION	PREPROCESSED LINES P_l = wc -l *.i	SOURCE LINES S_l = wc -l *.C	HEADER LINES $H_l = P_l - S_l$	% HEADER LINES
Perl	69958	23,214	46744	66
Class Library	177164	8372	168792	95
Ray Tracer	425717	18323	407394	95
CAD	443358	9114	434244	97
Web Browser	48920	1002	47918	97
Linker	1060368	36832	1023536	96
Optimizer	2979219	222367	2756852	92
Business Planner	279107	102597	176510	63
Average				87

Table 3

APPLICATION	COMPILE TIME OF PREPROCESSED LINES (SECS)	COMPILE TIME OF HEADER LINES (SECS)	% TIME SPENT COMPILING HEADERS
Perl	7.3	3.9	53
Class Library	17.2	17.1	72
Ray Tracer	44.7	38.1	85
CAD	38.7	31.3	80
Web Browser	14.6	14	95
Linker	85.8	77.6	90
Optimizer	194.2	146.5	75
Business Planner	70.5	34.6	49
Average			74

Precompiled headers (PCH) are a mechanism to cache a partially compiled version of one or more headers during a compile and then, reuse the cached version during subsequent compatible compiles. To elaborate, when the compiler is invoked on a source file *a.c* for the first time, the PCH mechanism snapshots the *#include* headers in a partially compiled state and preserves that intermediate form in a disk cache. Later, when *a.c* is edited and recompiled, the precompiled contents of the cache are simply ingested instead of recompiling the same headers again. This results in reducing a portion of compile time spent in processing the headers during the second and subsequent compiles.

There is, of course, the overhead of dumping the precompiled headers during the initial compile and the expense of ingesting them and reconstructing the compilation such that it mimics the actual reprocessing of the headers during the later compiles. In spite of this, in applications with large existing source bases that change incrementally, significant portions of the code remain dormant and are therefore prime candidates for PCH. In these cases the PCH scheme yields noticeable compile-time savings.

Several commercial compilers support some form of precompiled headers including Borland C/C++ *[5]* and IBM OS/390 C/C++ *[3]*. IBM C Set ++ *[4]* caches tokenized versions of headers while aCC's implementation caches headers in a partially compiled form that results after parsing and semantic analysis. KAI C++ *[2]* dumps and reads its raw symbol table contents while aCC serializes the symbol table and the intermediate form of the headers into the cache. Microsoft VC++ *[6]* offers automatic precompiled headers with a cache that is shared across a project and causes debug information to be emitted into every object file that uses the cache. This creates dependencies between the object files that share a cache and causes the compiler to rebuild all object files that use that cache if one changes. aCC's implementation does not share the PCH across source files. Other research directions for faster compiles include incremental compilation at a function level *[7]* and a compile server approach that retains internal data structures across compile requests *[8]*. Note that our paper is the only description of details that an implementation has to take care of.

Design Issues with Precompiled Headers

The basic idea of precompiled headers is simple: avoid reprocessing the *#include* headers each time a file is compiled by reusing a partially compiled version of the same headers from a cache. In the code sequence below, the compiler could simply cache the partially compiled contents of *a.h* and *b.h* in Example 1 into a single PCH cache for later use.

Example 1

```
// Start a.c
#include "b.h"
// Other C/C++ code …
#include "a.h"
// Other C/C++ code …

int main () {
// More code…
}
// End a.c
```

However, complications arise due to language and compilation system features that impose constraints on the save and reuse mechanisms. Such constraints impact the extent and contents of the region that can be precompiled and control the conditions under which the precompiled cache can be reused. The main design constraints are due to the:

- scope of the macro language in C/C++
- effect of external compile environment
- flexibility of header files

The C *[11]* and C++ *[10]* languages support a globally scoped macro preprocessor language that can penetrate any lexically succeeding header file boundary. This means that the contents of header files can be manipulated through macro symbols *external* to them. For instance, given the following contents of *a.h,*

Example 2

```
// Start a.h
#ifdef HPUX11
typedef size_t unsigned long
#else
typedef size_t unsigned int
#endif

struct List {
// More code…
};

#ifdef VER2
// More code…
#endif
// End a.h
```

and a source file *a.c,* that *#include*-s *a.h* as follows,

```
// Start a.c
#include "b.h"  ←May change a.h!
#define HPUX11  ←Changes a.h!
#include "a.h"
#define VER2 ←No effect on a.h


int main () {
// More code …
}
// End a.c
```

if the macro HPUX11 is defined or undefined in Example 1, the contents of *a.h* as visible to *a.c* is affected. Therefore, in order for *a.h* to be cached and reused correctly, either its partially compiled state should reflect the presence of the macro conditional in it, or the cached version should be sensitive to the lexically preceding macro settings in the source file that influence its contents. If not, the cached version of the header file is inapplicable in contexts with different macro settings.

Further, although macros that lexically succeed the *#include* statement do not exert any influence on its contents, C/C++ compilation systems do allow for macros to be set and unset through command-line options to the compiler. For instance, the *HPUX11* macro in *a.h* above could be set or unset through:

```
aCC -c -DHPUX11 a.c
        or
aCC -c -UHPUX11 a.c
```

Such macro settings affect the contents of all dependent header files and hence the contents of the precompiled cache. For a rigorous quantification of the incidences of macros, analysis of their usage patterns and their impact on development tools see *[1]*.

In addition to the compiler options that flag macros, options that control optimization levels and debug-information generation affect the generated code and hence impact the precompiled cache contents. For instance, in a non-debug-mode compile, the partially processed state for *a.h* may not have debug-information annotations for the *List* data-type and hence, neither will its precompiled cache. In a later compile, if the debug option is set and this cache is reused, no debug-information will be emitted for *List.* This is both unexpected and inconsistent with the current invocation of the compiler. Similar incompatibilities arise with options that control 32/64-bit code generation, exception-handling etc. So, for the correct reuse of the cache, the PCH mechanism must be aware of any compiler option settings that violate its consistency.

The above discussion shows that the PCH cache is affected by more than merely the headers' contents themselves. In fact, since macros preceding the *#include* statements can affect the contents of those header files, other lexically preceding header files can also toggle these macro values and hence alter header file contents. For instance, in Example 1, *b.h* is included before *a.h* and hence can affect the contents of *a.h* by changing the settings of macros that *a.h* depends on. This implies that if the user edits *a.c* to include *b.h* after *a.h* instead of before, the existing precompiled headers cache is rendered useless! In essence, the order of inclusion of header files important to the reuse of the header cache.

Also, many C/C++ language implementations support a variety of *pragmas* that control object layout, alignment, optimizations, inlining and code-generation. These, of course, apply to lexically following code, perhaps including header files. Furthermore, since header files in C/C++ are not required to be insular entities whose contents are complete, correct and guaranteed to compile in a stand-alone fashion, header files that compile in one context may fail in others. For instance,

Example 3

```
// Start b.c
static
#include "c.h"

extern void foo (int);
int main () {
foo (I);
}
// End b.c

// Start c.h
int I;
// End c.h
```

the code in Example 3 may be stylistically deplorable but it is perfectly legal. This implies that *#inlude*-d header files contents are highly context sensitive and impacted substantially by preceding text.

Implementation Issues with Precompiled Headers

Based on the previous discussion, in order to correctly capture the header files included in a given source file in a partially compiled state, an implementation must identify two things. First, it must demarcate the region to precompile, called the *passive region*, and then it must identify the parameters that influence the contents of this region, called the *configuration.*

Once these two entities are identified, they can be recorded in the cache when the source file is compiled. When a recompile is initiated by the user or the build system due to say, source-file changes or compile environment changes, the PCH mechanism must check to see if the *passive region* is untouched in the source file. It must also verify the compatibility of the *configuration* that controls the contents of the *passive region.* If these two attributes are intact, the cache can be reused for a faster compile.

The *passive region* is loosely described as a set of header file includes and it's preceding code, which impacts their contents. In identifying the start of this region, recall from the previous discussion that any source statements that precede the header files impacts their contents. Given that C/C++ source files are typically laid out with a series of header file include statements at the top, the *passive region* could simply begin at the head of the source file. In finding the end of this region, recall from the previous discussion that header file include statements can occur anywhere in the source file.

This implies that the end of the *passive region* could potentially coincide with the end of the source file causing the cache to contain the whole contents of the source file. Ostensibly, such a cache would derive maximal compile-time savings from reuse. In reality, such a cache would result in no savings at all! If the source file source is edited and recompiled, a cache that encompasses the gamut of the source file is automatically rendered void since no matter what part of the source file or its headers are altered, the cache is affected and cannot be reused.

So, the key to identifying the end of the *passive region* is to recognize that the overriding objective of the PCH mechanism is to reduce compile-times not by enlarging the volume of the PCH cache but by increasing the chances of its reuse. For existing programs with a largely frozen interface, the application header files seldom change and the system header files are immutable by the user.

Given this, a *passive region* that terminates with the first non-preprocessor non-comment statement in the source file is likely to cover any introductory comments, initial macro defines and conditional compile statements abetting the header file includes, but little else. This ensures that the *passive region* is not too trivial to yield benefits from precompiling. It also ensures that the *passive region* is limited and does not overrun the implementation in the source file that is prone to change more frequently. Example 4 illustrates the *passive region* boundaries in sample code.

Example 4

```
// Start foo.c
/* … */

        ← Start passive-region
#include <iostream.h>
#include <new.h>

#ifndef __GNU__
#define MIN(a,b) ((a<b)?a:b);
#endif

#ifdef __HPUX__
#include "hp_stack.h"
#else
#include "stack.h"
#endif
#include "foo.h"
         ← End passive-region

extern void foo (void);
int lookup () {
foo ();
// More code…
}
// End foo.c
```

A source file needs to be recompiled either because its contents changed or the contents of one or more of the directly or indirectly included headers changed or because the compile environment (e.g. command-line options, compiler version) changed. This threefold collection of parameters that impacts the contents of the *passive region* is called its *configuration* and the PCH mechanism must predicate the reuse of the header cache upon the compatibility of the *configuration* including that of the *passive region.*

The first *configuration* parameter is the contents of the source file. A change to the source-file contents could either lie within the *passive region* or outside it. Since the *passive region* begins at the top of the file, any alteration outside that region of code must be lexically beyond its scope and influence. This means that a source file change either directly touches the *passive region,* by say, adding or removing header file include statements or macro settings in it, hence violating the validity of the cache, or is extraneous to the *passive* *region* and does not prevents cache reuse. By recording the *passive region* itself, in addition to saving its precompiled form, the PCH mechanism can compare it against the version in the source file as a deciding factor for cache reuse during a later compile.

The second *configuration* parameter is the contents of all the direct and indirect header file dependencies of the source file. If the user edits one or more of the application headers to augment or alter its contents and one or more of these headers are included in the *passive region,* this nullifies the contents of the header cache. Therefore, in addition to a copy of the *passive region,* the header cache must contain the time-stamps of all dependent header files for the source file. This can then be checked by the PCH mechanism during a recompile to decide if the cache is worthy of reuse.

The third *configuration* parameter is the environment of the initial compile of the source file. This includes compilation command-line options, the compiler and/or PCH version and perhaps the OS versions and the compile location. Note that some compiler options like say, the verbose option, may be benign even if flipped and the partially compiled forms of the parse trees of various compiler versions may be in fact be compatible. However, for the sake of simplicity, the PCH mechanism can safely record all these parameters during the initial compile and check their consistency during a recompile.

The process that automates the decision making for creating and reusing the PCH can be implemented as follows. When the compiler is invoked on a source-file it checks to see if the corresponding PCH exists. If not, it proceeds to create one along with a record of its *configuration* parameters. If the PCH exists but its *configuration* is mismatched with the current invocation, then the compiler pretends that the PCH is absent and generates one. If a valid PCH exists, the compiler simply absorbs it, skips past the *passive region* and continues the compile.

In aCC's implementation, when it is first invoked on a source-file, it checks for the presence of a matching header cache. If no such exists, it slips into the *create* mode, in which it prepares to dump a PCH file with its associated *configuration.* First, it assembles a *configuration-record (CR)* that encapsulates the entire compile environment of the source file and its dependencies. The *CR* contains the following,

- the source filename
- the aCC version number
- the aCC command-line with all options
- header-file dependencies
- the time-stamps of the dependencies.

The OS version is not needed since aCC's version number uniquely identifies the major OS streams. In fact, the compiler version number serves to version the PCH, ensuring that the internal structure and format of the header cache are consistent with the expectations of the compiler i.e. what is written into the cache is what is read. In addition, it also captures a representation of the source in the *passive region.*

Note that the compile directory is not deemed necessary since the chances of erroneous cache reuse with eponymous files in two distinct locations are minimal. This is because aCC's implementation stores and verifies the *passive region* and the names and time-stamps of the dependencies before the reuse anyway.

aCC starts the *passive region* at the first token in the source file past any comments and white-space characters. In its search for the end of the *passive region,* it then skips past the preprocessing directives until it reaches the first declaration in the source file. aCC then pre-compiles the portion of the program within the *passive region* and dumps it into an eponymous header cache with its *CR*. It also stores a copy of the actual source code from the *passive region* in the *CR*.

In determining the end of the *passive region,* if the sentinel declaration occurs inside a conditional compile block, aCC raises the end of the *passive region* above the conditional compile block. Similarly, if aCC spots a header-file include statement inside a nested scope like a class or function, it stops the *passive region* in the file-scope prior to the start of the nested scope. This is shown in the code fragments below in Examples 5 and 6.

This is because, during subsequent recompiles of the source-file when the header cache is reused, aCC pretends that the source-file begins after the *passive region.* In this mode, called the *use* mode, aCC skips past the *passive region* to start the compile and the contents of the header cache are paged in only as needed. Tokenizing and parsing of the source-file begin immediately following the *passive region* and this requires the aCC scanner and parser to encounter legal start tokens in an outermost scope just beyond the *passive region.* In effect, the source-file without the *passive region* is expected to be a syntactically correct entity.

Example 5

```
// Start foo.c

        ← Start passive-region
#include <iostream.h>
#ifdef __HPUX__
#include "hp_stack.h"
#else
#include "stack.h"
#endif
#include "foo.h"
        ← End passive-region

#ifndef VERSION10
extern void bar ();
#endif
// End foo.c
```

Example 6

```
// Start foo.c
        ← Start passive-region

#include <iostream.h>
#include "foo.h"
        ← End passive-region

struct Tree {
#include "bar.h"
// More code…
};
// End foo.c
```

In addition, aCC supports a *pragma* that can be inserted by the user to define the end of the *passive region.* Such fine-grained control is convenient, for instance, when the user knows that a specific header file is currently in flux and therefore not amenable to precompile and wants to assert this to the compiler. In the absence of such a manual control, the automatic stop-point of the *passive region* may fall beyond the header in flux and hence negate any benefits that accrue due to the rest of the *passive region* preceding it.

Clearly, the manual stop-point must lexically precede the automatic stop-point to have effect. Of course, for reasons already discussed, the start of the *passive region* must always fall at the top of the file and hence is not open to user manipulation. Comments at the head of the source-file are not included in the *passive region* since they are semantically immaterial.

In *create mode,* once the *passive region* is identified and the *CR* assembled, the next step is to dump a partially compiled version of the headers into the cache. In aCC, partially analyzed parse-trees decorated with their symbol and type attributes are serialized to populate the precompiled header database during the *create mode.*

In *use mode*, once the configuration parameters mentioned above have been validated, the linearized parse-trees are read back from the disk cache on demand with only the cache preamble read in at start of the compile. As the compile progresses in *use mode*, if a required symbol or type is missing, the lookup fails and thereby triggers the PCH load mechanism for that symbol or type. Then, the symbol and its attributes are read in from the cache to fill in the in-core symbol table.

Results and Correctness

Table 4 presents the results of the automatic precompiled header implementation in aCC for complete end-to-end builds. The Web Browser, CAD and Business Planner are application subsets while the rest are complete applications. This implementation is entirely Makefile transparent and can be enabled and disabled through options or an environment variable.

As shown below, the average compile-time speed-up for these applications is over 2x. Perl has the least compile-time gains and the Web-Browser subset, the most. This matches the expectations set by Tables 2 and 3. Perl has the smallest ratio of header contents to source-file contents and the least time spent in compiling its headers while the Web Browser subset has a substantial portion of its code in its headers and it consumes a major fraction of its compile-time. Note that the cache is managed by aCC and not Make; it is therefore reused based on its validity or recreated by aCC. If the user chooses to explicitly delete the whole cache before the build, there are obviously no compile-time gains.

Table 5 juxtaposes the compile-time gains due to precompiled headers against the cost of the initially warming the cache via the *Use Factor.* The *Use Factor* is calculated using the equation:

$$nt_1 = t_2 + (n - 1)t_3$$

where t_1 is the compile-time without using pre-compiled headers, t_2 is the time to create the header cache and t_3 is the time to compile using the header cache. The *Use Factor (n)* then estimates the minimum number of times a file must be recompiled using its PCH to recover the cost of creating the cache.

Table 4

APPLICATION	COMPILE TIME OF PRE-PROCESSED SOURCE (t_1) (SECS)	COMPILE TIME FOR PCH USE MODE (t_2) (SECS)	SPEED-UP (t_1 / t_2)	% SPEED-UP
Perl	7.3	5.7	1.2	21
Class Library	17.2	7.4	2.3	56
Ray Tracer	44.7	19.1	2.3	57
CAD	38.7	18	2.1	53
Web Browser	14.6	3	4.8	79
Linker	85.8	21.1	4.0	75
Optimizer	194.2	76.9	2.5	60
Business Planner	70.5	43.8	1.6	37
Average			2.6	54

Table 5

APPLICATION	COMPILE TIME OF PRE-PROCESSED SOURCE (t_1) (SECS)	COMPILE TIME FOR PCH CREATE MODE (t_2) (SECS)	COMPILE TIME FOR PCH USE MODE (t_3) (SECS)	USE FACTOR (*n*)
Perl	7.3	11.3	5.7	3.5
Class Library	17.2	22.8	7.4	1.5
Ray Tracer	44.7	66.7	19.1	1.8
CAD	38.7	56.1	18	1.8
Web Browser	14.6	16.2	3	1.1
Linker	85.8	124.9	21.1	1.6
Optimizer	194.2	263.7	76.9	1.5
Business Planner	70.5	78.5	43.8	1.2
Average				1.5

Table 6

APPLICATION	SIZE OF OBJECT FILES (KB)	SIZE OF HEADER CACHE (KB)	RATIO OF CACHE SIZE TO OBJECTS SIZE
Perl	508.1	4699.9	9
Class Library	644.2	10324.8	16
Ray Tracer	1338.8	30714.2	22
CAD	1440.0	21490.6	14
Web Browser	392.4	3424.9	8
Linker	2346.9	61472.5	26
Optimizer	9644.9	107103.5	11
Business Planner	4855.6	12900.8	2
Average			13.5

Table 6 displays the size of the object files versus the size of the precompiled header cache for each of these applications. Since the cache occupies a significant amount of disk space, aCC provides an option to redirect it to a different disk or directory other than the default source file directory. This option can also be used to maintain multiple precompiled header caches, each for a different build target like say, debug build or optimized build.

In compiling these real applications, some correctness related issues surfaced due to:

- synthesized header files
- *__DATE__* and *__TIME__* macros
- revision control systems

One application recreated a set header files afresh for each build. This dynamic generation of header files at build-time effectively thwarted aCC's precompiled header mechanism. The generated header file time-stamps were always more recent than those recorded in the cache *CR* and hence would negate the cache on each build! This problem caused aCC to perpetually stagnate in the *create mode*, thus penalizing the application compile time further through the cost of creating the cache.

On examining this closely, it emerged that in actuality the generated header files were materially unaltered each time. That is, the contents of these header files did not change each time they were synthesized although their time-stamps did. Hence, it was clear that the header cache was in essence reusable since its contents were valid. In order to establish that, the cache *CR* was augmented to contain the checksum of the header dependency in addition to its name and time-stamp. aCC would now compute the checksum based on the contents of each header and include it in the *CR* along its names and time-stamps.

This technique also addressed the problem of time-stamp comparisons in the presence of multiple machines with possibly non-synchronous clocks. Further, in environments that upgrade to new system releases, the system and library headers' time-stamps are changed even when there are no material changes to them. Capturing the checksum in addition to the dependency time-stamp handles this situation as well.

Another application used the *__DATE__* and *__TIME__* macros in header files that were deemed potential hazards in using precompiled headers. While the C/C++ languages promise that the *__DATE__* and *__TIME__* macros will reflect the date and time during the translation of the source file, they do not dictate when that particular instance of translation must terminate. To elaborate, a compiler could start translating a source file with an occurrence of the *__TIME__* macro at 5.00 AM while compiling another such occurrence in the same file at 5.05 AM. Similarly, precompiled headers with occurrences of such macros could reflect the date and time during their translation, which may be different from when the rest of the source file is compiled or recompiled.

This behavior should be noted and clearly understood that when previously compiled components are integrated into a more recent set of compiles there may exist discrepancies in temporally sensitive code. One way to deflect the problem is to have the compiler abort the creation of the header cache for the current source file if it encounters these macros. This will result in a normal and correct compile but without the benefits of PCH. Another alternative is to accord special treatment to these macros. Typically, in aCC, the definitions of macros are stored in the header cache verbatim but their uses in the headers being precompiled are recorded after the macro is substituted. These two macros could be saved in their original form and hence recalled and replaced when the actual recompile happens, thus reflecting a more current date and time. aCC simply acknowledges this but does not yet implement this alternative in its precompiled header mechanism.

Some precompiled header implementations are based on the premise that source files in an application share a common set of headers that incur repeated processing in the course of building the application. Therefore, by factoring out the commonality into a precompiled form, the compiler saves on redundant processing between source files in addition to repeated processing within them. This scheme may also result in smaller header cache sizes but may require the user to reorganize the sources to *#include* a common set of headers in a specific order that can then be precompiled into a shared database. However, aCC's scheme does not share precompiled headers between source files and this quality in turn, enhances its usability in common application build environments.

User applications often rely on source code control and versioning software to select, build and maintain correct versions of their source files. One such commonly used system is *Clearcase [9]*. *Clearcase* maintains clear correspondences between source file elements, their dependencies and the object files that derive from them. By creating a header cache that is common to several source files, a new constraint is introduced on all the object files that are derived from each of those sources.

For example, say that the source file *a.c* containing header files *a.h* and *foo.h* produces a header cache that is shared by *b.c* since it also included the same headers. *Clearcase* then records this cache as a dependency for both *a.o* and *b.o*. Later, say *a.c* is edited to remove *a.h* and recompiled, then the cache is deemed invalid for reuse with *a.c* and the compiler enters the *create mode.* It rewrites the cache and creates a new object file, *a.o.* Since that cache has been marked as a dependency for *b.o,* it triggers a rebuild of *b.c.* This renders the same cache unusable for *b.c* since it now contains different headers and recreates it! This cross dependency can confuse *Clearcase* and lead to a deadly cycle of rebuilds that immobilizes productivity.

aCC's implementation of the header caching scheme attaches a unique PCH to every source file, thus eliminating any dependency between the cache and multiple object files. Note that the overriding goal in this of PCH is to favor the correctness of results over maximizing the reuse of the PCH. This implementation thus trades-off any savings in disk space that may accrue from sharing the PCH cache across many source files for improved usability with real application development environments. This may be considered a cost-effective trade-off since the effects of increased disk consumption may be mitigated to some extent through cache compression.

Not including the compile directory in the *CR* also allows various users compiling the same file from different directories to reuse the object file and cache. This does not compromise the correctness of this scheme since over and above the *CR*, the *passive region* must also match.

Other applications mimic source-code control systems in a limited fashion through *view-pathing. View-pathing* is a feature that delineates a set of paths for the master-copies of certain header files while reserving another set for holding the user-modified versions. Through the flip of a switch the user can select the preferred version over the original version.

For example, let *a.c* include *a.h* and *a.h* reside in the directory *sysinc* during an initial compile. Let the user now copy *a.h* to another directory, say *userinc*, and edit it. If the user now recompiles the file, the *passive region*, the *CR* and the originally recorded set of dependencies are proven valid and the cache is reused. However, the desired alternate header file is not selected!

```
#Create cache
aCC -c -Isysinc -I- -Iuserinc +hdr_cache a.c

#User copies a.h and modifies it. Use cache!
aCC -c -Isysinc -I- -Iuserinc +hdr_cache a.c
```

aCC's implementation of precompiled headers is insensitive to *view-pathing* and hence is blind to the additions of header file to alternate locations in the list of *-I* directories. One possible remedy may be to augment the *CR* with the *-I* directories' contents and detect any changes in these prior to using the header cache.

Conclusion

An automatic header caching mechanism has been implemented in aCC and is available in its currently shipping versions. The full implementation including the load-dump mechanism (not discussed in this paper) consists of about 14,000 lines of commented C++ code. This exercise helped identify several key design and implementation issues due to the language definition and existing compile environments in providing a usable and fully automatic precompiled header mechanism. This paper discusses many of these issues, describes aCC's implementation choices and trade-offs and suggests remedies to open issues. The advantages of this scheme are:

- Ease of use. It requires no manual intervention. Source files need not be tailored to contain *#include*-s in any specific order. Header files need not be altered to have include guards around them.
- It is transparent to the Make process and friendly to revision control systems

The biggest limitation is that the header cache consumes a significant amount of disk space. Finally, this paper presents the results of using aCC's PCH mechanism on a set of applications. It improves compile-times by over 50%.

References

[1] Michael Ernst, Greg Badros, David Notkin. *An Empirical Analysis of C Preprocessor Use*. Technical Report UW-CSE-97-04-06, Department of Computer Science and Engineering, University of Washington, Seattle.
[2] *KAI C++ ™ User's Guide, Precompiled Headers.* http://www.kai.com/C_plus_plus/v3.4/doc/UserGuide/precompiled-headers.html
[3] *IBM OS/390 V2R6.0 C/C++ User's Guide, Chapter 10 Using Precompiled Headers.* http://www.s390.ibm.com:80/bookmgr-cgi/bookmgr.cmd/BOOKS/CBCOUG03
[4] *IBM C Set ++ for AIX*. http://www-4.ibm.com/software/ad/caix/
[5] C. Horstmann. *An in-depth look as Borland C++*. C++ Report, 3(10), pp. 17-20, Nov-Dec 1991.
[6] *Microsoft VC++ Language Help, Automatic Use of Precompiled Headers*.
[7] Fyfe, Soleimanipour, Tatkar. *Compiling from Saved State: Fast Incremental Compilation with Traditional; Unix Compilers.* Usenix, Winter 1991.
[8] T Onodera. *Reducing Compilation time by a Compilation Server*. Software Practice and Experience. Vol 23(5), May 1993.
[9] Rational Software Corporation, *Clearcase* http://www.rational.com/products/clearcase/
[10] *ANSI/ISO C++ Final Draft International Standard*
[11] *ANSI C Standard*, ANSI ISO/IEC 9899:1990
[12] *aCC : The HP ANSI C++ Compiler.* http://www.hp.com/go/hpc++