Reducing Build Time Through Precompilations for Evolving Large Software

Yijun Yu
University of Toronto

Homayoun Dayani-Fard
IBM Canada

John Mylopoulos
University of Toronto

Periklis Andritsos
University of Toronto

Abstract

Large-scale legacy programs take long time to compile, thereby hampering productivity. This paper presents algorithms that reduce compilation time by analyzing syntactic dependencies in fine-grain program units, and by removing redundancies as well as false dependencies. These algorithms are combined with parallel compilation techniques (compiler farms, compiler caches), to further reduce build time. We demonstrate through experiments their effectiveness in achieving significant speedup for both fresh and incremental builds.

1 Introduction

Managing complexity of large-scale software development is central to software engineering [30]. Software systems, under maintenance pressures for improved functionality, better quality and more services, are becoming more complex by the Lehman's second law of evolution [19]. Such pressures obscure the internal structure and quality of the software, making it difficult to understand and maintain it [5]. We studied an evolving large-scale C/C++ software within IBM over several releases and documented its various growth factors. Figure 1 depicts the number of program entities (broken down into functions, variables and types), number of source files, average number of included header files, and number of inter-dependencies among components. The number of program files in the software system has been growing steadily for the past four years, with jumps near major new releases. Similarly, the number of actual dependencies have grown as well as the average number of header files included by program files.

Typical large-scale C/C++ programs consist of many *compilation* and *header units*. A compilation unit (e.g. '.c' file) will be compiled into an object file [18]. It is also called a *translation unit* in the GCC community [12]. A header unit (e.g. '.h' file) will be included into a compilation unit prior to compilation. At a small granularity, we define a *program unit* (PU) as a declaration or definition of a symbol. User-defined symbols must be *defined* once globally in the program, but can be *declared* multiple times in different compilation units. In C/C++, global variables and functions are regarded as definition units; the remaining symbols are declarations, such as static functions/variables, function/variable prototypes, classes, structs, unions, typedefs, enumerators, etc. Most declarations are grouped into the headers and *preprocessed* into the compilation unit by replacing the `#include` directives with the content of corresponding headers. A full expansion of the `#include` directives results in a *preprocessed image*.

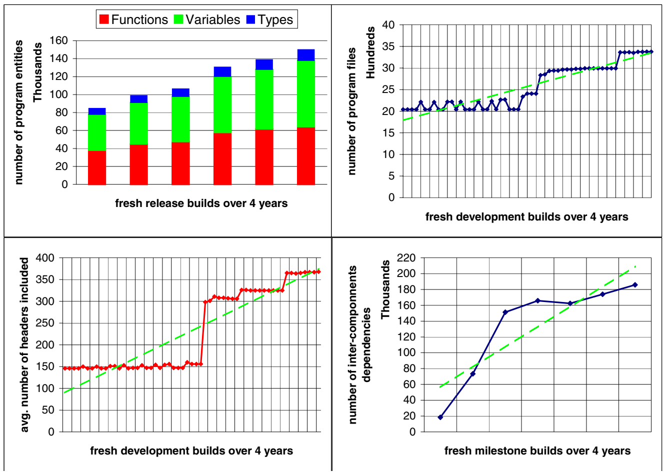


Figure 1. The growth of an industrial software

Introducing headers generally reduces *redundancies* (i.e. duplicate declarations) to save space and reduce update inconsistencies. However, headers can be included by multiple files and as such may contain declarations that are *falsely* (i.e. not) needed by some compilation units that include them. Although the functionality of a system is not affected, *redundancies* and *false dependencies* in the preprocessed compilation units do affect the efficiency of the development process: the longer the build process (e.g., compilation and linking) takes, the longer developers have to wait in order to integrate their changes.

This paper presents a *precompilation* (i.e. source-to-source transformation before compilation) approach to the removal of redundancies inside compilation units and false dependencies among the files. We show that removing declaration redundancies within compilation units can speedup the *fresh* builds (i.e., compiling everything from scratch), while removing false dependencies can speedup the *incremental* builds (i.e. compiling only the changes). Unlike file level dependency checking, our approach analyzes the dependencies among *fine-grain* program units inside headers and compilation units. Furthermore, to reduce the overhead of using the exact dependencies, we adopt an approach that needs *light-weight* amount of information to be extracted. The resulting precompilation technique is transparent to the build process, incremental to the development and independent of the choice of compiler. In addition to the algorithms for identifying and removing redundancies and false dependencies, the paper also presents an experimental evaluation.

The rest of the paper is organized as follows. Section 2 presents definitions and algorithms that serve as foundations for our approach. Section 3 outlines experimental results when applied to a public-domain software (VIM [21]) in C as well as an industrial component in C++. Section 4 describes and compares related work in the compilation optimization area. Section 5 provides some concluding remarks.

2 Our approach

Our process consists of four steps: 1) the compilation units are parsed by an adapted GCC 3.4.0 parser into sequences of program units; 2) a redundancy removal algorithm is applied to remove unnecessary PUs by traversing the abstract syntax tree (AST) once; 3) a partitioning algorithm is performed on the lexically ordered necessary PUs to regroup them into headers and compilation unit source files, while preserving their dependency order; 4) finally, a logical grouping of files is created by clustering of the generated compilation units to reduce the number of generated headers. In the remainder of this section we present each step of our approach in detail.

Extracting program units A unit u is *lexically* before unit v, if u occurs before v in one of the preprocessed images. A program unit u *depends on* another program unit v if u uses v and v occurs lexically before u.

The program unit extraction is an algorithm implemented by adapting the GCC compiler: given a compilation unit as a sequence of tokens (the lexicon stream) from the lexical analyzer, the algorithm converts it into a sequence of program units. The token strings are concatenated into a character stream, which is recorded upon the identification of a program unit and reset to empty for the next token. This allows us to accurately locate program units by their start/end lines and columns.

Based on the stream, the GCC parser constructs an abstract syntax tree T. During the parsing, we are interested in the tree nodes that may be identified as program units. The `-fdump-translation-unit` option in GCC is not sufficient for this purpose because external references are not stored in the abstract syntax tree. Thus we adapted the GCC type-checker that calls the `build_external_ref` routine to keep track of the dependency of the program unit on the externally referenced declaration unit. The output of one compilation unit is a sequence of program units P. Each program unit in P has an associated code segment and at least one node in T. On the other hand, each declaration node corresponds to at most one program unit. The new option in our adapted GCC is called `-fdump-program-unit`.

We illustrate the parsing process by an example compilation unit, which is dissected into a sequence of 7 program units. Each has a character stream until the next program unit. An alias shown in the comment is associated with the kind and the name of a program unit. E.g., “struct:A” indicates a “struct” with a name “A”.

```
typedef int NUMBER;          //PU@1 type:NUMBER
struct node;                 //PU@2 forward:node
typedef struct node {        //PU@3 struct:node
 float value;                //
 struct node* next;          //   <- PU@3,PU@2
} *list;                     //    type:list
struct A {                   //PU@4 struct:A
 union {                     //
   NUMBER value;             //        <- PU@1
 } u;                        //     union:u
};                           //
extern int                   //
printf(char *format,...);    //PU@5 funcdcl:printf
enum {                       //PU@6 enum:<anonymous>
  Satisfied,                 // enumerator:Satisfied
  Denied,                    // enumerator:Denied
};                           //
int main(argc, argv)         //PU@7 funcdef:main
int argc; char **argv;       //
{                            //
  list l, n;                 //        <- PU@3
  for (n = l; n; n=n->next)  //
    printf("%f", n->value);  //        <- PU@5
  return (int) Satisfied;    //        <- PU@6
}                            //
```

Several aliases may refer to the same program unit because they do not separate from each other, e.g. the `struct:node` and `type:list` are aliases to `PU@4`. Aliases help the parser to link partial information, such as the enumerator constant `Satisfied`, to the declaration of the anonymous `enum` type.

Entities such as field names, parameter names and auto variables are not considered program units because they are not needed for parsing other program units. E.g., `union:u` inside `struct:A` is not considered a program unit. In this manner, much fewer entities need to be recorded compared to the traditional fact extraction.

The dependencies are extracted for symbols that were previously identified as a program unit, e.g. the PU "7" depends on 3 previously defined PU's $\{3, 5, 6\}$. The output is thus a lexically ordered sequence of program units.

A compiler such as GCC 3.4.0 can have 36 phases from parsing source code into generating hardware instructions. We stop right after the first parsing phase using the option `-fsyntax-only` to have a quicker precompilation, while the other phases will be called in the compilation phase of the precompiled code.

Removing redundancies Among all the program units P, we denote the set of *definitions* C as program units that will be stored in the object file, while the set of *declarations* is defined as $H = P \setminus C$ [18]. A *program unit dependency graph* (PUDG) is a digraph $G(P, D)$, where the vertexes in P represent program units, and the edges in $D \subset H \times P$ represent the dependencies among PUs. Given a lexical order $\prec$, the set of program units in a compilation unit is converted into a sequence P, where $P[i]$ denotes the i-th program unit in the sequence. Now a *light-weight* PUDG (LPUDG) is defined as a digraph $G^{\prec} = (P, \prec)$ implied by the lexical ordering of the sequence $\prec$: $P[i] \prec P[j] \iff i < j$. As not all pairs of the program units in the sequence have dependency between each other, $D \subset \prec$.

For each compilation unit, we keep a sequence of program units (which implies LPUDG) rather than storing the complete PUDG. We will show that having the seemingly less accurate LPUDG is enough for the redundancy removal and also enough for the header restructuring.

In the set of declarations H, only a subset $N \subset H$ is necessary for the correct compilation of C, while other declarations $R = H \setminus N$ can be removed. After dependency extraction, the immediately dependent declarations for the i-th program unit $P[i]$ can be identified as $N(i)$. Thus the necessary declarations N are the union of all the declarations that are transitively depended by C. This is done through traversing the extracted PU sequence twice. Initially all the definition units are marked as necessary. Then the first traversal iterates through the PU sequence backward from the end to the beginning, marks all the PU's that are directly depended by a currently necessary unit as necessary. After the traversal, all necessary declarations N and defintions C are marked. The second traversal simply outputs the marked PU's from the beginning to the end. Among a parsed sequence of 7 program units in our example program, H=$\{$1–6$\}$ are declarations, $C = \{7\}$ is the only definition. Then based on the extracted dependencies, the backward traversal marks $\{3, 5, 6\}$ as necessary for 7, then skips the dependency $1 \rightarrow 4$, and marks $\{2\}$ as necessary for 3. At the end of the traversal, program units $R = \{1, 4\}$ will be removed because the definition in $C = \{7\}$ does not depend on them transitively. Unlike this example, the density $((|N| + |C|)/|P|)$ of necessary elements in the real applications tends to be much smaller. The reason is that most system headers contain redundant declarations for different platforms. For example, a single line of declaration for `printf` is necessary in the complete 843 lines of the `stdio.h` if it were included, which still contains 406 LOC after removing blank lines produced from the #ifdef macros by the command `cpp -E -P`.

Removing false dependencies If the same program unit will be included multiple times in different compilation units, it is better to place it into a constructed header. We have presented elsewhere [31] an algorithm to remove redundancies and false code dependencies based on the heavy weight PUDG extraction. In this paper, we adapted GCC for the program units extraction and the efficient redundancy removal. The new header restructuring algorithm only requires the necessary program unit sequences obtained from individual compilation units.

Given the containing relation between files (headers and compilation units) and program units, we define a *file dependency graph* (FDG) $\mathcal{G} = (\mathcal{F}, \mathcal{D})$ where $\mathcal{F}$ represents the set of files. Each element of $\mathcal{F}$ contains a subset of program units P in the PUDG. The vertices $\mathcal{F}$ are separated into headers $\mathcal{H}$ and compilation units $\mathcal{C}$, then the edges $\mathcal{D} \subset \mathcal{H} \times \mathcal{F}$ are the dependencies. The relation between the PUDG G and the FDG $\mathcal{G}$ is determined by the N-to-1 partitioning mapping $\mathcal{X} : N \times \mathcal{F}$, where each element of $\mathcal{F}$ is a partitioned (disjoint) subset of N.

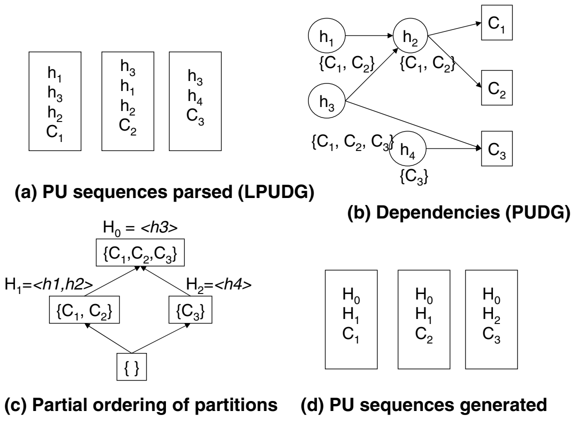
In a file dependency graph, a dependency between a file with program units $A \subset P$ and a file with program units $B \subset P$ is *false* if there is no PU dependency from any PU $a \in A$ to any PU $b \in B$. In header restructuring, we only consider false dependencies caused by spurious PUs in headers. A remedy to this problem is to split the header so that only true dependencies occur.

We do not split the compilation units as individual definitions because the false dependencies in the compilation units have no impact on the build time. Therefore we keep the existing mapping between definitions and compilation units and replace all definitions C^i in the i-th compilation unit with one node $i \in \mathcal{C}$. Thus the new PUDG have a vertex set $N \cup \mathcal{C}$ where N is the union of the necessary declarations in all the compilation units $\mathcal{C}$.

Each necessary declaration $u \in N^i$ of the i-th compilation unit has a dependency (u, i) in the new LPUDG. After the union of the global declarations in compilation units, we also know a set of compilation units that depends on each $u \in N$: $\mathcal{D}(u) = \{i | u \in N^i\}$. Starting from a naive partitioning where each declaration in H is a separate partition set, we merge the PUs that belong to the same sets of compilation units and update the partitioning $\mathcal{H}$. The resulting partitioning describes the headers to be generated. Now we present the pseudo code of our restructuring algorithm.

Algorithm 1. Header Restructuring

```
Input: The sequences of necessary declarations $N^i$ for each
    compilation unit $i \in \mathcal{C}$;
Output: A partition of $N$ where $N = \bigcup_{i \in C} N^i$ and generated
    header and compilation units.
begin /* Initializing */ $N = \{\}$; $\forall u \in N : \mathcal{D}(u) = \{\}$;
  for each $i \in \mathcal{C} : N = N \cup N^i; \forall u \in N^i : \mathcal{D}(u) = \mathcal{D}(u) \cup \{i\}$;
  /* Partitioning */ let $\mathcal{H} = \{\{u\} | u \in N\}$;
  repeat done = true;
    for each $A, B \in \mathcal{H}$ and $A \neq B$:
      if $\bigcup_{a \in A} \mathcal{D}(a) = \bigcup_{b \in B} \mathcal{D}(b)$ then:
        $\mathcal{H} = \mathcal{H} \cup \{A \cup B\} \setminus \{A, B\}$; done = false;
        for each $k \in \bigcup_{a \in A} \mathcal{D}(a) : \mathcal{H}^k = \mathcal{H}^k \cup \{A \cup B\} \setminus \{A, B\}$;
  until done;
  for $k = 1, |\mathcal{H}|$ : PrintSortUnits($k$, $\mathcal{H}^k$, ComparePUs);
  for $i = 1, |\mathcal{C}|$ : PrintSortUnits($i$, $N^i$, $\mathcal{H}$, ComparePU);
end
ComparePUs(I: program unit sets $A$, $B$: $A \neq B$; O: $\prec, \succ, or =$ )
begin if $\mathcal{D}(A) \supset \mathcal{D}(B)$ return $\prec$;
        if $\mathcal{D}(A) \subset \mathcal{D}(B)$ return $\succ$;
        return $=$; end
ComparePU(I: program units $a$, $b$; O: $\prec, =$ , or $\succ$ )
begin if $\exists k : \mathcal{P}^k[i] = a \wedge \mathcal{P}^k[j] = b$ return $i - j$;
        return $a - b$; end
```



(a) PU sequences parsed (LPUDG)

(b) Dependencies (PUDG)

(c) Partial ordering of partitions

(d) PU sequences generated

Figure 2. The illustrative steps of the header restructuring algorithm

When generating code with a set of program units $\mathcal{H}^k$, a proper order is used to sort them into a sequence. In our algorithm, the program units in a generated header are compared using their lexical order if there is a compilation unit in which both of them occur. The header units in a compilation unit are compared using a partial order defined by the set inclusion relationship between their transitive closures: $A < B$ iff $\mathcal{D}(A) \subset \mathcal{D}(B)$. Figure 2 illustrates how the algorithm uses the sequences of necessary program units to derive a set of headers and to generate the right sequence of header inclusions and definitions in the compilation units. The lexical ordering of program units (LPUDG) in (a) already implies the explicit dependencies among them (PUDG): the dependencies on declarations are found by clustering equivalent classes for each declaration in (b); the partial ordering of the equivalent class partitions is defined by the set inclusion relationship (c); and the header inclusions are generated by sorting with the partition ordering (d). The generated code may change the order of declarations. For example in Figure 2a, a sequence of declarations h_1, h_3, h_2 is restructured into a sequence of header inclusions H_0, H_1, where H_0, H_1 are generated headers with declarations h_3 and h_1, h_2 respectively (Figure 2c). Therefore the new compilation unit will have a new sequence of headers h_3, h_1, h_2 after inclusion expansion. By the properties below, we prove that the new sequence maintains the dependencies.

The following properties hold for the FDG $\mathcal{G}$ after header restructuring.

1. **No false dependencies in headers.** For any two declaration program units u, v in the same generated header file, if there is a dependency path from u to a compilation unit $w \in \mathcal{C}$, there is also a dependency path from v to w.

   Proof. There is a dependency path from $u \in N$ to $w \in \mathcal{C}$ i.f.f. $w \in \mathcal{D}(u)$. By the header partitioning procedure, if u, v are in the same generated header, then $\mathcal{D}(u) = \mathcal{D}(v)$. Thus $w \in \mathcal{D}(v)$, in other words, there is also a dependency path from v to w. □

2. **Largest granularity.** If a set containing two nodes from separate partitions is considered as a header, then a false dependency is introduced. In other words, For any two vertices u, v in two different generated headers, there is a $w \in \mathcal{C}$ such that either there is a path from u to w but no path from v to w, or there is a path from v to w but no path from u to w.

   Proof. Since u, v belong to different partitions, $\mathcal{D}(u) \neq \mathcal{D}(v)$. Either $\mathcal{D}(u) \cap \mathcal{D}(v) = \phi$ or $\mathcal{D}(u) \cap \mathcal{D}(v) \neq \phi$. If the intersection is empty, any node $w \in \mathcal{D}(u) \cup \mathcal{D}(v)$ satisfies the conclusion; if the intersection is not empty, then any node $w \in (\mathcal{D}(u) \setminus \mathcal{D}(v)) \cup (\mathcal{D}(v) \setminus \mathcal{D}(u)) = (\mathcal{D}(u) \cup \mathcal{D}(v)) \setminus (\mathcal{D}(u) \cap \mathcal{D}(v))$ satisfies the conclusion. □

3. **Correct generation of code.** The generated code respects the dependencies in the PUDG $G = (P, D)$. In other words, if there is a dependency between any two program units $a^i, b^i \in P, (a^i, b^i) \in D$ in the same compilation unit $i \in \mathcal{C}$, then they are generated in the

order of a^i, b^i in the restructured code after preprocessing.

Proof. (1) By calculation, *ComparePUs*(H^i, H^j) returns $i < j$ if $\mathcal{D}(H^i) \supset \mathcal{D}(H^j)$ and returns $i \neq j$ if $\mathcal{D}(H^i) \neq \mathcal{D}(H^j)$. (2) Also, we prove that a dependency $(a, b) \in D$ implies $\mathcal{D}(a) \supseteq \mathcal{D}(b)$: Since we have removed redundant program units, thus for any compilation unit i where b occurs, there is a definition program unit $c^i \in N^i$ such that $(b, c^i) \in D^i$. By the transitive property of dependency relations, if $(a, b) \in D^i$, then $(a, c^i) \in D^i$. In other words, $i \in \mathcal{D}(a)$. Therefore $\mathcal{D}(a) \supseteq \mathcal{D}(b)$. This establishes a lattice (Figure 2(c)). (3) Now, consider a violation of the PUDG dependency happens as $(b, a) \in D$ while $a \prec b$. If both a, b are program units in the same file, they must follow the order in the original sequence of program units by *ComparePU*, thus $b \prec a$; if b is in a generated header and a is in the definition part of the compilation unit, then $b \prec a$ by the output order of the algorithm; if they are included from two different generated headers: $a \in H^i, b \in H^j$ and $i \neq j$, since $a \prec b$ then $i < j$. However, by (1) and (2), $i \geq j$. Therefore, in any case, the violating dependency does not occur between two outputted program units in the lexical order defined by the algorithm. □

Given these properties, it is not necessary to place the duplicate inclusion guards around any generated header since they are included only once in each compilation unit. The time complexity of the partitioning procedure is $\mathcal{O}(|N| \times |\mathcal{C}|)$ operations.

The restructured header inclusions after false dependency removal help to accurately identify the compilation units that need to recompile as a program unit is changed, without wasting time on the compilation of falsely dependent compilation units. Therefore, header restructuring can reduce the time spent on incremental build at the expense of increasing number of generated headers.

Directory clustering Having partitioned the program units into headers, one can think of an additional optimization to partition the headers and associated compilation units together into a local directory.

Given the partitioned compilation units, the directory restructuring moves the generated headers along with the compilation units to the corresponding directories. 1) Convert the FDG into a list of dependency vectors on the compilation units, similar to the output from the Intel compiler `-M` option or `makedepend`. 2) Pass them to a clustering algorithm to group similar records together. As a clustering algorithm in our approach, we use LIMBO [2], which minimizes the loss of information across the clusters it builds. The outcome from the algorithm is n partitions of the compilation units. We move the compilation units and the headers into n separate directories corresponding to the partitions. The samples provided to the clustering algorithm accurately reveal the architecture and guarantee a good clustering. 3) Next, factor out the common headers used by each directory until there is no more redundancies. To save time, we only compare the common headers among all directories (in this case 1), the common headers between two directories ($n(n-1)/2$) and the remaining distinct headers in each directory (n). Moreover, a factoring is performed on directories A and B only if $(|A \cap B|/|A \cup B|) > \epsilon$ where ϵ is a threshold with a default value of 0.5. 4) Finally preprocess the small headers in each directory into a large one and change the corresponding inclusions in every compilation unit.

This change to the FDG will introduce some false dependencies back into the program. It is a trade-off between componentizing the software system and introducing overhead for the incremental build [6].

3 Experiments

We have applied our approach to two case studies: VIM 6.2 and a mature industrial product owned by IBM (Figure 1). Our experimental results, presented in the following subsections, show that, in both cases, the improvement of our approach was significant relative to other approaches that do not use our optimizations.

3.1 Restructuring VIM

The public domain program VIM (Vi IMproved) 6.2 [21] is studied. The source code includes 61 `.c` files, 24 `.h` headers, 38 `.xpm` headers and 56 `.pro` headers[1]. The complete code base has 220 KLOC.

Prepare the code bases and the compilation farm. After running the `configure` command for Linux, `-g -O2` option and 49 compilation units were automatically chosen as *original* code base. These units depend on 355 unique headers. Transitively, each file on average includes 290 headers. The precompilation (`-fdump-program-unit`) and restructuring (`-fdump-headers`) were done automatically using our adapted GCC 3.4.0, while the LIMBO clustering algorithm was performed after the desired number of clusters was chosen as 3 according to the number of components in Model-View-Controller (MVC) architecture [16][2]. Apart from the original code base, we obtained three additional code bases, namely the *precompiled*, *restructured* and *componentized* ones.

[1] Additional headers will be introduced from the inclusion of system headers

[2] The resulting clusters do follow the MVC architecture for VIM.

Table 1. Header statistics.

	Original	Precompiled	Restructured	Componentized
Header	527,271	0	261,376	125,440
Compilation units	5,095,601	5,366,778	4,557,615	3,983,735
Total bytes	5,622,872	5,366,778	4,818,991	4,109,175
No. of unique headers	355	0	925	5
No. of header inclusions	14,276	0	5,778	138

In Table 1, the bytes needed for storing the code base and the number of inclusion directives are compared.

The experiments on VIM were carried out on a network of Linux workstations. The host machine for the compilations is a 2.20 GHz Intel Pentium 4 workstation, with 512 KB cache. The OS is RedHat Linux 7.3, with kernel version 2.4.20-30.7 compiled by GCC 2.96. We also used the servers available in the local area network of our campus lab. The compilation farm can use up to 8 processors: 2 x 2.8GHz, 4 x 2.4GHz, 1 x 2.2GHz (the local workstation) and 1 x 1.6GHz. All machines use the same operating system, although `distcc` allows for cross-compilation. The times are measured as the average of 10 separate runs of the same settings.

Improvement of fresh builds The average size of preprocessed files was reduced from 708.9 KB to 104.71 KB. The overall build size is reduced from 33.9 MB to 5.01 MB. The size saving comparisons of individual compilation units are shown in Figure 3a. The data items are horizontally sorted by the original preprocessed file size. The similar shapes of the two curves indicate that the reduction happens almost uniformly to every compilation unit. The time savings and their comparisons are shown in Figure 3b. The compilation time is almost uniformly reduced for each compilation unit, since almost every unit in VIM includes `vim.h`. The net speedup by precompilation is 251% (2.51x). The precompilation overhead is needed only for the first fresh build. Taking into account precompilation overhead for the first build, the time is still 12.6% faster than the original fresh build. If the precompiled code is compiled N times, then the overhead can be divided by N. The restructured and componentized code has less time reduction than fresh build.

We also compared the compilation time for the complete program when different compiler options were used. To compare with parallel compilation, we chose `distcc`, a distributed compiler specially designed for C/C++ compilation. Other parallel compilers are general purpose and we do not intend to compare with the C/C++ compilation. To compare with compilation cache `ccache`, we used the `$HOME/.ccache` as a shared directory for the cached files. The first run is after the cleanup of the cache using `ccache -C`, the second run with `ccache` is just after the first one to utilize the cache. The third and fourth runs are associated with different parallel `make` options `-j5` and `-j20` respectively. We also compared two different C/C++ compilers GCC 2.96 (gcc) and Intel C/C++ compiler 8.0 (icc) on the Linux system[3]. The time of the fresh build using different `make` options is shown in Figure 4.

When there are available processors, in our case 8, the parallel compilation applying `distcc` leads to a speedup of 3x (3 times), far below 8 because of the network traffic in the environment. When the compiler cache or preprocessed header technique is applied for the first time, the compilation degrades by warming up the cache; when they are applied later, their net speedup over parallel build was in the range of 60%. For example, the combined speedup becomes 5x after applying the caching techniques on top of `distcc`;

Our precompilation further reduces compilation time by 1.5x to 8x further than over techniques. The highest overall speedup 39.59x is reached when all the above techniques are combined for GCC compilation, while the highest net speedup using precompilation over other techniques is 8.41x. The net speedup is higher on a heavy-loaded compiler farm for parallel build than that for sequential build because the code size reduction also reduces the network bandwidth demands by sending/receiving preprocessed images to/from remote compilers. In summary, our redundancy removal precompilation is orthogonal to parallel compilation, compilation cache and precompiled headers techniques.

Improvement of incremental builds In this experiment, we estimate whether incremental builds can be improved. The change data of VIM at each incremental build is not available[4], therefore a probability analysis is used by assuming that a code base per incremental build has changed ΔL lines of code and the probability of change for each line is uniform $\Delta L/L$ where L is the total lines of code (LOC).

Consider a file dependency graph (FDG), and measure the line of code for each file as L_{H_i} for headers H_i or L_{C_i} for compilation units C_i. The probability of chang-

[3] Due to the platform chosen, we did not test the Microsoft Visual C/C++ compiler and the HP C/C++ compiler cited in the related work

[4] The publicly committed CVS log does not match the real development changes since not all changes were committed to the repository.

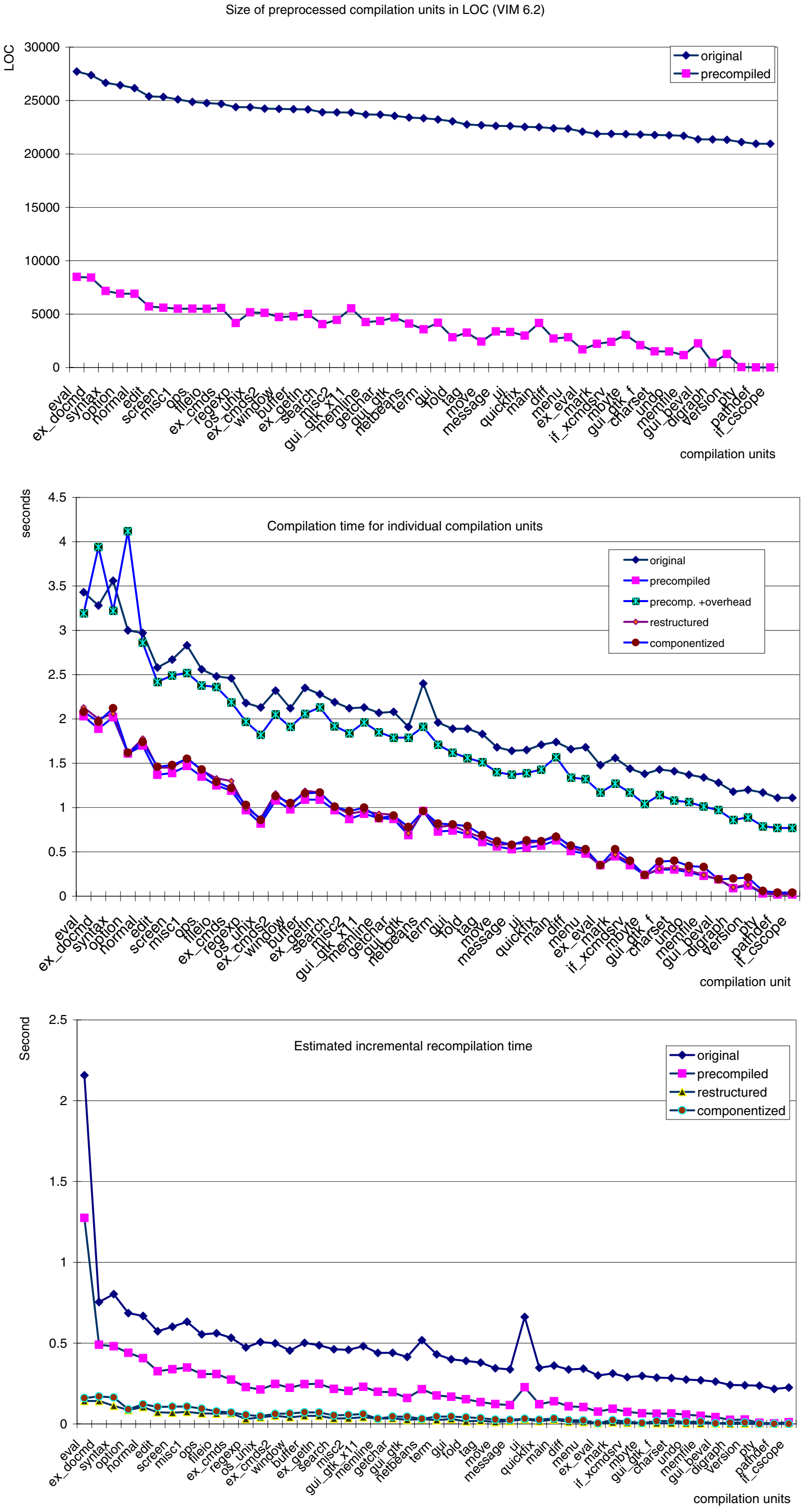


Figure 3. (a) Size, (b) Fresh build time, (c) Incremental build time for VIM code bases

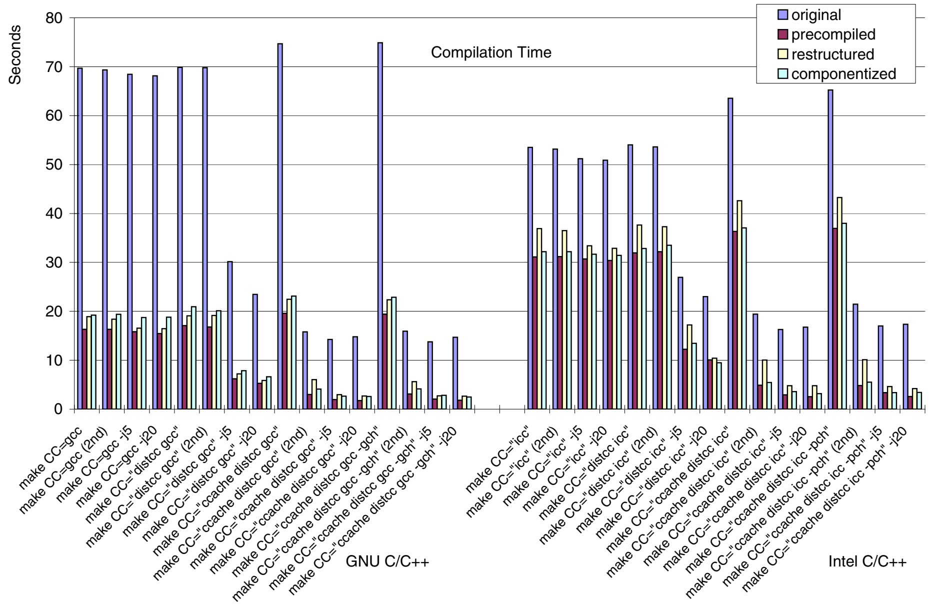


Figure 4. Fresh build time of the original, precompiled, restructured and componentized VIM.

ing a header H_i or a compilation unit C_i is $L_{H_i}\Delta L/L$ or $L_{C_i}\Delta L/L$ respectively. For each changed header H_i, one can expect all the dependent compilation units $\mathcal{D}(i)$ need a recompilation, whereas for each changed compilation unit, only itself will be recompiled. In the original code base, a compilation unit C_i needs a recompilation if either its implementation is changed, or any of its dependent headers is changed. If we measure the time for its recompilation as t_i, then the overall incremental build time is

$$\begin{array}{l}\Delta t = \sum_i p_i t_i \text{where} \\ p_i = [L(C_i) + \sum_{j|i\in\mathcal{D}(H_j)} L(H_j)]\Delta L/L\end{array} \qquad (1)$$

The precompiled code base uses the same FDG as the original, but Equation (1) is adjusted to Equation (2) since the directly changed compilation unit needs an overhead t'_i of redo the precompilation, while indirectly changed compilation unit can recompile quicker with the precompiled code to amortize the overhead.

$$\begin{array}{l}\Delta t = \sum_i [p^c_i(t_i + t'_i) + (1 - p^c_i)p^h_i t_i] \quad \text{where} \\ p^c_i = L(C_i)\Delta L/L \\ p^h_i = \sum_{j|i\in\mathcal{D}(H_j)} L(H_j)\Delta L/L\end{array} \qquad (2)$$

For the restructured and componentized code base, equation (1) is used, since the restructuring and clustering needs to be done only once before the incremental builds. Here, a smaller size parameter L and a reduced FDG were used.

Having the LOC of source files (in Figure 3a) and the timing of the compilation units (in Figure 3b), the incremental build analyses of the *original*, *precompiled*, *restructured* and *componentized* code bases are shown in Figure 3c: for each compilation unit, the estimated recompilation time per incremental build is calculated using Equation (1) for the *original*, *restructured* and *componentized* program or using Equation (2) for the *precompiled* program. In total, for the *original*, *precompiled*, *restructured* and *componentized* code base, an incremental build when changing one line of code takes respectively 22.73, 10.06, 1.76 and 2.46 seconds of recompilation (see Figure 3c), whereas their fresh build including linking takes 97.89, 39.04, 41.1 and 40.91 seconds respectively.

Testing We verified that both the header restructured and componentized VIM programs have the same functionalities as the original program using the 51 test cases accompanying the VIM source code under `testdir` directory. Among them, the original VIM succeeded in 49 test cases except for a test case for “gf” (case 17) and a test case “insert expansion” (case 32). According to documentation in VIM, they are designed for testing VIM on the WIN32 platform. It is worth noting that the restructured and the componentized VIM also succeeded in the 49 test cases and failed in the same 2 test cases.

3.2 Restructuring an industrial program

The overall gain in the build time, though significant, may not impact productivity significantly. For instance, in the VIM case study, the amount of activity and the absolute value of the build time is not large enough to justify the restructuring effort. To conduct a more realistic evaluation of our proposal, we applied it to shrink-wrap software product [5] with 112 components (organized as directory of files) and over 7 million lines of C++ code (Figure 1).

Over the years, the header dependencies in the codebase have decayed to the extent that each program file in our component, on average, includes 543 headers (directly or indirectly). The average size of each compilation unit is around 37 KB, expanding 53x to around 1.96 MB after inclusions. Though the component under study only has 172KLOC in its compilation units, or 2.8% of the system, the distinctly included headers have 20MB, almost 34.7% of all the distinct headers in the system. The average build time from scratch – including the preprocessing – for this component is around 19 minutes. This number reduces to around 14 minutes if the files are preprocessed. Applying precompilation alone, the preprocessed size reduction was 182.1MB, or 50.42%. The build time was 4.35 minutes, saving 9.57 minutes over preprocessed program.

4 Related work

A `Makefile` declares a set of dependency rules between targets [10]. Only the targets that transitively depends on a more recent target will be executed. The `make` *targets optimization* finds truly dependent targets and removes unnecessary ones in the transitive closure of the end target [11, 28]. Unless the code to compile is generated during the build, most compilations can be fully *parallelized* across different compilation units. Thus if the development machine has multiple processors, a '-jN' option for `make` can fork N processes to do the compilation tasks at the same time[6]. Using a network of workstations (NOW), `pmake` [7], `pvmmake` [8], `mpimake` [9], `dmake` [28], `lsmake` [24] and `distcc` [25] all aim at dispatching parallel compilation jobs to a set of workstations. In particular, `distcc` is a parallel C/C++ compilation tool that utilize the available workstations in a `compilation farm`.

Usually parallel compilation tool should work along with a *caching* mechanism for a compilation. It takes much more time for a compiler to *preprocess* an input file and expand them into a stream of text for parsing, than to load the preprocessed file directly. Thus the preprocessed file can be stored in a cache to speedup the preprocessing. Hashing the cached entries can help locate the preprocessed files stored in the cache even faster. `ccache` [29] implements the compiler cache by placing the preprocessed files into a directory where each of them are hashed and shared, similar technique [15, 23] does caching within a server compiler.

The *precompiled header* (PCH) option is implemented in modern C/C++ compilers to cache the compiled headers in order to reuse the compilation result[7]. Different from `ccache`, PCH techniques deal with headers only and the cached results are in object form, thus the cached headers can not be shared among different compilers. Unlike including all program units in the headers by the PCH approach, our header restructuring selectively includes the program units that are necessary for the compilation units. The program unit dependence graph is much finer than the file dependencies, thus lead to an additional speedup to the PCH. Our precompilation result is in source form, ready to share among different C/C++ compilers.

A concept related to false dependency is the Ratio of Use to Visibility (RUV) [4]. Here *Use* defines the number of compilation units where a declaration is used and *Visible* defines the number of compilation units where the declaration is used. RUV can be seen as an indicator of false dependencies. After our header restructuring, the ratio will be restored to 1. In [1], the cost to various recompilation techniques was surveyed. The *cascading* recompilation triggers recompilation whenever a change to the `make` target happens; the *surface* recompilation does not trigger a *cascading* recompilation when changes are made to comments; the *cutoff* recompilation triggers a *cascading* recompilation only when changes are made to preprocessed images. The *smart* recompilation in [1] triggers a *cascading* recompilation only when a change is made to the smallest file dependency graph derived from the headers. Unlike us, these techniques do not restructure the headers to reflect the true dependencies, rather it maintains a dependency graph using existing headers, thus the RUV they obtained was still below 1; the link-time *smartest recompilation* has to rely on the type inference to generalize types of undeclared identifiers, and as noted by the author, it may be counterproductive because it slows the error removal.

Elsewhere [31] we presented an algorithm to remove all false dependencies through header restructuring. The algorithm relies on 3rd party parsers (also called *fact extractors*) such as CPPX [14], Datrix [3, 13], KLOCwork [26] or the `-fdump-translation-unit` option in GCC [12], to prepare a cumbersome *abstract syntax graph*, which records all the direct symbol relations in a relational tuple format. In [31], we developed a dependency extractor based on the array of detailed entities generated from one of the specialized fact extractors. The speed of the *heavy-weight dependency extraction* was slow due to the large number of excessive entities and relations extracted. For exam-

[5] Due to confidentiality issues, we cannot disclose the software name.

[6] Even on a single processor system, forking two parallel tasks usually outperforms sequential make because the CPU can be better utilized rather than waiting for the I/O devices.

[7] Commercial compilers have implemented precompiled headers as an advanced option, for example: Microsoft Visual C/C++'s `/YX` option [22] generates precompiled headers as `*.pch` files, Intel C/C++ compiler's `-pch` option [27] generates `*.pchi` files. So does HP ANSI C/C++ compiler [17] and an NeXT implementation [20] where a detail explanation the mechanism in the precompilation can be found. Starting from version 3.4.0, GNU GCC can also precompile headers into `*.gch` files [12].

ple, during the compilation of VIM 6.2 (Section 3), we obtained 72,056 various dependencies among 22,489 program units, whereas the fact extractor in Datrix would report 3,008,664 various relations among 1,852,326 entities. With the same objective to remove redundancies and false dependencies, this paper reports efficient algorithms to extract a sequence of program units along with parsing implemented in our adapted GCC compiler. Comparing with the explicit program unit dependency graph, the program units sequence has less complexity (lighter-weight) and result in an efficient precompilation and header restructuring. In addition, the precompiled or restructured code base are smaller than the precompiled headers as well as the preprocessed files. Our precompilation results are compiler-independent since the results are in source form and can be reused by other C/C++ compilers. The adapted GCC compiler is also transparent to the *make* process as the precompilation is implemented into a `-fdump-program-unit -fsyntax-only` option and the header restructuring as an additional `-fdump-headers` option.

5 Conclusion

This paper presented a set of algorithms and techniques for improving the speed of compilation in large scale C/C++ programs. The need for such an approach was driven by a study of large scale industrial programs. The presented algorithms rely on syntactic dependencies and can be used as a precompilation. Our experminents have shown that this technique is orthogonal to other optimization techniques such as parallel compilation, caching and precompiled headers. Apart from improving the fresh build process, we also presented a light-weight fine-grain header restructuring technique that can achieve efficient incremental builds. The overhead of preprocessing the generated headers can be further reduced by a clustering-based componentization. By adapting the GCC compiler to include our precompilation as its options, no change is needed on the existing `Makefile`. Experiments showed that it can achieve up to 8 times gain over the speed of compilation already tuned with parallelism and locality and a large-scale C/C++ software can apply this precompilation technique.

References

[1] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *TOSEM*, 3(1):3–28, Jan. 1994.

[2] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *10th WCRE*, pages 334–344, Nov. 2003.

[3] Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). Technical report, Bell Canada, 2000.

[4] E. A. Borison. *Program Changes and the Cost of Selective Recompilation.* PhD thesis, Carnegie Mellon University, 1989.

[5] H. Dayani-Fard. *Quality-based software release management*. PhD thesis, Queen's University, 2003.

[6] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the build architecture of legacy C/C++ software systems. In *FASE 2005*, pages 96–110.

[7] A. de Boor. Pmake - a parallel make. Technical report, U.C. Berkeley, Fall 1987.

[8] J. Devaney, R. Lipman, M. Lo, W. Mitchell, M. Edwards, and C. Clark. PADE - the parallel applications development environment. Gaithersburg, Maryland 20899, 1995.

[9] J. E. Devaney. Converting pvmmake to mpimake under LAM, and MPI and parallel genetic programming. In A. Lumsdaine, editor, *MPI Developers Conference*, 22-23 June 1995.

[10] S. Feldman. Make - a program for maintaining computer programs. *SPE*, pages 255–265, April 1979.

[11] C. J. Fleckenstein and D. Hemmendinger. A parallel 'make' utility based on Linda's tuple-space. In *17th ACM conference on Comp. Sci.*, pages 216–220. ACM Press, 1989.

[12] GNU. http://gcc.gnu.org/gcc-3.4/.

[13] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *WCRE*, October 1998.

[14] R. C. Holt, A. E. Hassan, B. Lague, S. Lapierre, and C. Leduc. E/R schema for the Datrix C/C++/Java exchange format. In *WCRE*, pages 284–286, 2000.

[15] B. Koehler and R. N. Horspool. CCC: A caching compiler for C. *SPE*, 27(2):155–165, 1997.

[16] G. E. Krasner and S.T.Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of OOP*, 1(3):26–49, 1988.

[17] T. Krishnaswamy. Automatic precompiled headers: Speeding up C++ application build times. In *WIESS'2000 in conjunction with USENIX OSDI'2000*. ACM, 2000.

[18] J. Lakos. *Large-scale C++ software design.* Addison-Wesley, 1996.

[19] M. M. Lehman. Laws of software evolution revisited. *Lecture Notes in Computer Science*, 1149:108–120, 1996.

[20] A. Litman. An implementation of precompiled headers. *SPE*, 23(3):341–350, Mar. 1993.

[21] B. Moolenaar. Vim 6.2. In *http://www.vim.org*, 2003.

[22] MSDN. Visual C++ precompiled header compiler options.

[23] T. Onodera. Reducing compilation time by a compilation server. *SPE*, 23(5):477–485, May 1993.

[24] Platform, Inc. Using `lsmake`. In *LSF User's Guide*.

[25] M. Pool. distcc: a fast, free distributed C/C++ compiler. In *http://distcc.samba.org*.

[26] N. Rajala, D. Campara, and N. Mansurov. InSight: reverse engineer case tool. In *ICSE*, pages 630–633, 1999.

[27] D. Schouten, X. Tian, A. Bik, and M. Girkar. Inside the Intel compiler. *Linux Journal*, 2003(106):4, 2003.

[28] Sun Microsystems. Distributed make: http://wwws.sun.com/software/sundev/news/features/dmake.html.

[29] A. Tridgell. ccache: http://ccache.samba.org.

[30] H. van Vliet. *Software Engineering: principles and practice, 2nd Ed.* John Wiley, 2000.

[31] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software development processes. In *CASCON'03*, pages 288–297, Oct. 2003.