# Removing False Code Dependencies to Speedup Software Build Processes

Yijun Yu, Homy Dayani-Fard, John Mylopoulos
{yijun,jm}@cs.toronto.edu, homy@ca.ibm.com
University of Toronto, IBM Toronto Lab

## Abstract

The development of large software systems involves a continual lengthy build process that may include preprocessing, compilation and linking of tens of thousands of source code files. In many cases, much of this build time is wasted due to false dependencies between implementation files and their respective header files. We present a graph algorithm and a programming tool that discovers and removes false dependencies between files. We show experimentally that the resulting preprocessed code is more compact, thereby contributing to faster build processes.

## 1 Introduction

Large software systems typically consist of a number of source files. These codes include headers (e.g., `*.h`) and implementations (e.g., `*.c`). The header files contain the necessary program units for the implementation files. Such units, in C, include external function declarations, global variable declarations, structure definitions, enumerations and type definitions. By including header files through preprocessor directives (e.g., `#include`), an implementation file creates *dependencies* on other implementation files (or libraries) that facilitate the use of the program units in the included headers. In an ideal scenario, an implementation file includes only the necessary definitions and declarations that it will use. However, due to lack of information and supporting tools, as a software system evolves, these dependencies grow and result in excessive redundant inclusions. While optimizing compilers can remove such redundancies from the executable binary image, the duration of the building process can drastically increase due to the size of the preprocessed files.

While excessive inclusions and false dependencies do not affect the functionality of a system, they affect the efficiency of the development process and result in the waste of resources. Furthermore, a false dependency between an implementation file and its header can cause unnecessary compilation of the implementation file when an independent part of the header has changed. Consequently the incremental build time is also prolonged by false dependences. Hence, the efficiency of the build process can be affected both during incremental build (i.e. compiling what has changed) as well as fresh build (i.e. compiling everything from scratch). Considering the nightly build paradigms [3] the overall effect of false dependencies can add up rapidly.

Traditional approaches to improving the efficiency of the build processes focus on removing false target dependencies in make files. These approaches do not consider the internal details of implementation files , however, this paper presents a novel approach to the removal of false dependencies based on analyzing the header files. The approach is incremental and

can be applied at various levels of complexity (e.g. function declaration only or global variables only) based on the non-functional requirements. The main steps of this approach involve:

1. constructing a dependence graph of file dependencies;
2. partitioning the dependence graph to remove false dependencies between files; and
3. reorganizing the header files to reduce coupling between files and improve cohesion.

The rest of the paper is organized as follows: section 2 describes the refactoring algorithms for discovering and removing false dependencies between files; section 3 demonstrates the application of the refactoring algorithm to an example program; section 4 shows result of applying the tool to a public-domain software VIM 6.1 [11]; section 5 compares the presented approach with related work; section 6 discusses the future work; and section 7 provides some concluding remark as the potential application of this tool.

# 2 Refactoring software dependencies

Software refactoring [7] is a process of applying a sequence of small non-functional changes to a software system to improve its overall quality. To effectively manage software quality, we must manage the code-base health as well as the overall quality of the end product. One aspect of the code-base health in this study involved file dependencies [4]. As the code-base evolves, the number of false dependencies grow rapidly, which contributes to the longer time for building of the product and potential loss of component (e.g. a group of related files) interfaces.

To reduce the dependencies among different files, Makefile optimization can be performed to discover target dependencies: a set of files that depend on others. However, this approach tolerates false dependencies at the code level, which can contribute to timely builds. An increase in the build time, in turn, can reduce the availability of the product for testing and other quality assurance tasks.

The loss of component independence is another effect of false code dependencies. When the dependent component is outside the scope of the developer, she will have to wait for the whole system to be built before a thorough integration test can be done. To overcome the test problem, code stubs of the interfacing modules are created to replace the dependent component in order to allow earlier unit tests before the whole system is built. Code stubs are a work-around to tolerate the false code dependencies for speeding up the development process, but a final integration test has to be performed to demonstrate their smooth integration with the rest of the system.

The approach proposed in this paper differs from the above mentioned in that it relies on fine-grain source code dependence checks. This approach analyzes the dependencies at the program units level to minimize build time.

## 2.1 Exposing false code dependencies

The first step in identifying false dependencies involves the construction of a dependence graph. Using parsing technologies (e.g. CPPX [9] or Datrix [2]), a set of relations can be extracted from the source files. These relations determine in which file a program unit entity (e.g. a function, type, or variable) is defined and where it has been used. Furthermore, these relations provide the exact location (i.e. line number) where the program unit is defined or used. The dependence relations between the define and the uses of a program unit form a graph, where the program unit define/use instances form the set of vertices and the edges connect the corresponding define and uses vertices. We call this graph a *code dependence graph* to differentiate its semantics from the program (data/control) dependence graph [6].

Formally, a code dependence graph is a digraph [1] $G = (V, E)$ where the vertices in $V$ represent the defines/declares of program units, and the edges in $E \subset V \times V$ represent the dependencies between program units. The vertices are further divided into two mutually exclusive sets: $V = V_H \cup V_C$. $V_H$ represents

the set of program units that should be placed in header files such as declarations of variables and functions, and definitions of types, structures and enumerations. $V_C$ represents the set of program units that should be placed in the implementation files such as definition of variables and functions. For example, consider the two files `main.c` and its included header `foo.h` as shown below:

```
--main.c--
#include ‘‘foo.h’’
int main () {
    foo ();
}
--foo.h--
void foo ();
void bar ();
```

The set $V_H$ has two vertices `foo_declare` and `bar_declare` and the set $V_C$ has one vertex `foo_define`. The only edge in this graph connects the two vertices declaring and defining the function `foo`.

In a complete system, a program unit that occurs both in $V_H$ and $V_C$ causes a *true* dependencies between them. If a header file contains two program units $u, v \in V_H$, where $w \in V_C$ depends on $u$ but not on $v$, then a *false* dependence $(v, w)$ forms. Having false dependencies implies not only potential increasing of preprocessed file sizes, but also an unnecessary rebuilding of any files containing $w$ any time $v$ is modified.

To construct a code dependence graph, we use the `Datrix` parser [2]. `Datrix` constructs an Abstract Syntax Graph (ASG) expressed in Tuple Attributes format (TA [8]). Using the data from the ASG, we construct the code dependence graph. Using the code dependence graph, we remove the unused program units for the definitions in the program, i.e., remove all nodes in $V_H$ that have an empty dominate set in $V_C$. Then we apply the code partitioning algorithm.

## 2.2 Code partitioning algorithm

Given a code dependence graph $G = (V, E)$, all false dependencies can be removed by putting each element of $V_H$ in a separate header file. This trivial partitioning has undesired side-effects: the code is scattered into $|V_H|$ header files that blur the big picture of the data structures. A better partition condenses related definitions into a smaller number of headers while avoiding false dependencies. As a result, we need to devise a non-trivial partitioning algorithm to obtain the largest partitions granularity without false dependencies.

Before partitioning, we preprocess the graph to determine the influence of a header file over an array of the implementation files. For each vertex in the sets $V_H$ and $V_C$, we first calculate two transitive closures as *Dominates* $D$ and *Dominated* $D'$ defined as:

1. **for each** $u \in V_H$
2. $D(u) = \{v | (u', v) \in E, u' \in D(u), v \in V_C\}$
3. **for each** $v \in V_C$
4. $D'(v) = \{u | (u, v') \in E, v' \in D'(v), v \in V_H\}$

These calculations require $O(|V|)$ transitive closure operations if all vertices are calculated separately. We can avoid this complexity by finding a spanning tree of the graph, then calculate the dominate set of a vertex as the union of all its child vertices. Thus less than $O(|V|)$ set union operations are required.

Next, we merge the vertices which belong to the same *Dominates* or the same *Dominated* sets:

1. **for each** $u, v \in V$
2. **if** $u, v \in V_H$ **and** D(u)=D(v)
3. **or** $u, v \in V_C$ **and** D'(u)=D'(v)**then**
4. $condense(u, v)$
5. **end if**

The complexity of the partitioning procedure is $O(|V|^2)$ set comparison operation plus less than $O(|E|)$ condense operations. Here $condense(u, v)$ is a procedure that removes the vertex $v$ while replacing any edges $(v, w)$ by $(u, w)$ and $(w, v)$ by $(w, u)$ as long as $w \neq u$. When $w = u$, the edges $(u, v)$ and $(v, u)$ are removed to prevent cycles:

**Input:** $u, v \in V_H \cup V_C$
1. **for each** $e \in E$
2. **if** $e.to = v$ **and** $e.from \neq u$ **then**
3. $E = E \setminus \{e\} \cup \{(e.from, u)\}$
4. **else if** $e.from = v$ **and** $e.to \neq u$ **then**
5. $E = E \setminus \{e\} \cup \{(u, e.from)\}$

6. **else if** $e = (u, v)$ **or** $e = (v, u)$ **then**
7. $\quad E = E \setminus \{e\}$
8. **end if**
9. $V_H = V_H \setminus v$

The complexity of the condense operation is $O(|E|)$. Therefore, the above partitioning procedure costs $O(|V|^2+|E|^2)$ operations. As a result, the partitioning algorithm groups not only header files, but also the implementation files. The granularity is as large as possible while false dependencies are fully removed, stated as follows.

Given the digraph $G = (V, E)$ where $V = V_H \cup V_C$, the algorithm obtains a condensed graph $G' = (V', E')$ where $V' = V'_H \cup V'_C$ and $V'_H, V'_C$ are partitions of $V_H, V_C$ respectively. The following properties hold for the graph $G'$:

1. **No false dependencies.** For any two vertices $u, v$ in $u' \subset V_H$ where $u' \in V'_H$, if there is a path from $u$ to $w \in V_C$, then there is also a path from $v$ to $w$; for any two vertices $u, v$ in $u' \subset V_C$ where $u' \in V'_C$, if there is a path from $w \in V_H$ to $u$, then there is also a path from $w$ to $v$.

2. **Largest granularity.** For any two vertices $u \in u'$ and $v \in v'$ where $u', v' \in V'_H$, there is a $w \in V_C$ 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 $w$.

In practice, it is not desirable to partition the implementation files because the generated code may become unfamiliar to the developers. Thus we provide a second mechanism to impose a constraint on the graph partitioning that all the function definitions in one file are still in the same file. Therefore we desire to remove all the false dependencies between different files, and tolerate the false dependencies within the files.

For this purpose, the partitioning algorithm can still be applied, with an exception that $V'_H = \{f \mid \text{file}(u, f), u \in V_H\}$ is used instead of $V_H$ and $E' = \{(\text{file}(u), \text{file}(v)) | u, v \in V\}$ is used for $E$ where $\text{file}(u, f)$ is the relationship between a program unit $u$ and a filename $f$. If $u \in V_C$, then let $\text{file}(u) = u$.

The same algorithm also applies to still larger granular modules such as components, which can be considered as the abstraction of a collection of files. Generally, the larger the granularity, the more false dependencies are tolerated.

## 2.3 Generating the code

The code generation for the target system is based on the location information stored with the program units. A program unit is associated with an original source program file name and the beginning and ending line numbers. So once the partitions are calculated, the corresponding code segments can be extracted from the original source program. Along with the moved code, we attach a line `# line file` indicating the source of the program unit so that the developer can still see where they are taken from. The names of the implementation files are the same as the original implementation files, but the names of new header files are generated by a sequence number.

For each partition, the program units must be output in the topological order of the dependence graph. However, the topological ordering requires that the graph be acyclic. In our code dependence graph, such cycles do exist. For example, the following code segment can produce a cycle between definitions of a `typedef` unit and a `struct` unit:

```
typedef struct list list_T; struct list {
   int key;
   list_T * next;
};
```

We break a cycle $a \leftrightarrow b$, e.g. `struct list` as $a$ and `typedef list_T` as $b$, by removing $a \rightarrow b$ while for each vertex $c$ in the graph such that $c \rightarrow a$, we add an edge $c \rightarrow b$ to make sure all the uses of $a$ and $b$ in $c$ still follow the definitions of $a$ and $b$ in the generated code. For independent vertices, we generate the code in the same lexicographical order as they were in the original code for the programmer to recognize the code in relation to the original code.

Having the partitions of header files and implementation files, one can think of an additional optimization to move the header files and associated implementation files together into a local directory. As long as other implementation files do not refer to the header file that was removed, such a restructuring can improve

the cohesion of the system[1]. This process is done by checking the number of implementation files a header serves. If this number is one, we move the header file to the corresponding module directory in order to increase component cohesion.

The next step in the code generation is to inline the included files. An header file with single out-degree will be embedded into the file that includes it. In this way even fewer new header files will be generated. Finally a Makefile is generated with the new file names to avoid manual intervention.

# 3 An example

To illustrate the code partitioning algorithm, this section provides a simple example. The subject program has the following file structures:

```
./
  include/
    header.h    # declares a, b, ..., i
  module1/
    1.c         # defines a, b, c
    2.c         # defines d, e, f, g
  module2/
    3.c         # defines h, i
  main.c        # defines main
```

`header.h` contains declarations for 9 functions $a, b, c, d, e, f, g, h, i$. Every `.c` file includes `header.h` in order to use the external functions. However, as it can be seen, not all included function declarations are necessary. The preprocessed size of the implementation files can be obtained by running a preprocessor (e.g. `gcc -E -P`). The above example has 69 lines of code. The following list shows the caller-callee relation between functions.

```
main -> f, g, h, i
d -> a
f -> b
g -> c
i -> c, d
h -> e
```

[1]This restructuring creates a similar structure as Java packages.

Using the above list, the partitioning algorithm produces the following code structure which also satisfies file-scale granularity constraints and localizing requirements

```
./
  include/
    2.h         # declares c
  module1/
    1.h         # declares a, b
    1.c         # no inclusion
    2.c         # include 1.h, 2.h
  module2/
    3.h         # declares d, e
    3.c         # include 2.h, 3.h
  0.h           # declares f, g, h, i
  main.c        # include 0.h
```

After the reorganization, the size of the preprocessed files is reduced to 42.

In practice, size reduction can be larger. This is due to inclusion of the entire library header files for a small number of program units. For example, most implementation files include the entire `<stdio.h>` while using only one or two functions, e.g. `printf`. Our algorithm can reduce the preprocessed file size dramatically by leaving out unnecessary program units.

# 4 Experiments

The header refactoring system is shown in figure 1. First, the C implementation files are compiled with `gcc` (step 1). Next, running the preprocessor `gcc -E -P` we measure the size of the resulting files (step 2). The preprocessed files are parsed by `Datrix` and passing the program units for the header refactoring to be carried out (step 3). The new program files are compiled once again to assure that previous steps did not remove any necessary dependence (step 4). Finally, we measure the new code size using the preprocessor only(step 5).

For large systems, the refactoring of multiple files can be done incrementally by replacing some of the object files with the refactored ones. This facilitates the incremental build of the system. Developers can perform the refactoring of their own headers without waiting for the whole system to be ready.

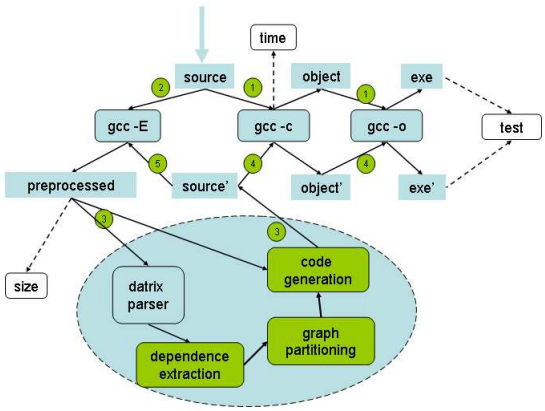


Figure 1: The overview of the header refactoring system. The usage is shown in steps: (1) compiling the original program; (2) measuring the original code size; (3) refactoring headers (4) compiling the result program; (5) measuring the result program.

We have applied the header optimization on a public domain software system VIM (Vi IMproved) 6.1 [11]. We consider the source code of version 6.1 which includes 65 `.c` implementation files, 28 `.h` header files and 56 `.pro` header files generated for function declaration prototypes using `cproto`. A successful compilation on the Linux platform requires 47 `.c` files.

To demonstrate the idea of the incremental build, we just refactor 15 of the 47 files, and link the new object files with the other 32 to ensure that the program compiles. Table 1 lists the result of refactoring in terms of lines of code, number of words, and number of bytes before and after using word count (`wc`).

The refactored file size include the #pragma which says where the program units are extracted from. If these pragma's are removed, there are more reductions. In summary, the reduction of build size in bytes is 89.70% for the incremental build and 26.38% for the fresh build.

Table 2 shows the build time gain after refactoring. Initially we compare the build time using the preprocessed files and the refactored files. This comparison shows a 39.58% reduction for an incremental build and a 10.70% reduction for a fresh build. The preprocessed files, however, are larger than the original files because all inclusions are included. For example, the `buffer.c` explodes from 108916 bytes to 365106 bytes after preprocessing, and shrinks back to 134138 bytes after refactoring the headers, and shrinks further to 113861 bytes by removing the unnecessary `#` pragma's. Therefore we shall compare the compilation time between all the four sets of equivalent program files.

A developer would choose a normal compilation for the incremental build, and choose an optimizing compilation for the fresh build before the release of the software. Therefore time for both the normal `gcc` build and the optimized `gcc -O3` build were measured. It is interesting to see that the absolute time gain for both build are about the same (1.34 vs. 1.28 sec.), which indicates that the large build size reduction helps mostly for reducing syntax parsing time while the other optimization process time are not much influenced. Therefore we may expect that the time reduction ratio for the fresh build is smaller than that for the incremental build. Since the software development is a process of frequently rebuilding the code, the normal time reduction ratio 32.24% for incremental build matters to the development time.

# 5 Related work

Our approach is compared with existing tools and related topics.

## 5.1 Existing tools

Existing tools such as `make` and `makedepend` are commonly used to maintain the code dependencies and speedup the build process by avoiding the unnecessary compilation of files that were not modified. However, the build-time dependencies have larger granularity in comparison to variables and functions at the program unit level. For example, every implementation `.c` file of VIM includes `vim.h`, which in turn includes almost every exported functions from the `*.pro` files. That means a change to a single `.c` file requires almost all other files to be rebuilt. Having a function prototype extractor like `cproto` helps creating headers, but the use of them can potentially slow down the build of

Table 1: Measuring the size of the original and the refactored VIM program files after `gcc -E -P`. Refactoring 15/47 of the source files leads to about 26.38% reduction of total size in bytes.

| Size | # lines | # lines' | # words | # words' | # bytes | # bytes' |
|---|---|---|---|---|---|---|
| buffer.c | 23531 | 4155 | 64414 | 11463 | 724131 | 112228 |
| charset.c | 21485 | 1345 | 58625 | 3933 | 660718 | 32354 |
| diff.c | 22108 | 2488 | 60471 | 6584 | 683455 | 63393 |
| digraph.c | 21072 | 448 | 57307 | 1458 | 651256 | 11590 |
| edit.c | 24929 | 5120 | 68136 | 14847 | 767021 | 148998 |
| fold.c | 22770 | 2663 | 62257 | 7771 | 701359 | 74801 |
| mark.c | 21601 | 2019 | 59742 | 5777 | 672967 | 52969 |
| memfile.c | 21411 | 1277 | 58302 | 3383 | 660857 | 31604 |
| memline.c | 23396 | 3862 | 64748 | 11016 | 733120 | 115647 |
| misc1.c | 24606 | 4932 | 67487 | 14424 | 755816 | 141171 |
| ops.c | 24434 | 4981 | 67385 | 14184 | 757182 | 142253 |
| pty.c | 20824 | 42 | 56711 | 139 | 645466 | 1067 |
| tag.c | 22396 | 2938 | 61232 | 7819 | 699744 | 83291 |
| undo.c | 21479 | 1335 | 58541 | 3598 | 665507 | 35988 |
| version.c | 21024 | 1331 | 57090 | 2699 | 651200 | 26684 |
| subtotal | 337066 | 38936 | 922448 | 109095 | 10429799 | 1074038 |
| reduction | | -88.45% | | -88.17% | | -89.70% |
| other 32 files | 743885 | 743885 | 2356587 | 2356587 | 25033807 | 25033807 |
| total | 1080951 | 782821 | 3279035 | 2465682 | 35463606 | 26107845 |
| reduction | | -27.58% | | -24.80% | | -26.38% |

the whole program. Other techniques such as pre-compiled headers help reducing the time for file inclusions, however, they do not remove the redundant program units except for macros.

To extract finer granularity code dependencies, a code factor extractor based on a parser is necessary. Existing parsers such as `CPPX`, `Datrix` [9] extract facts from a C program and store them in usable formats such as TA [8] and GXL [10]. Among them, `CPPX` reports program units relations including macros, while `Datrix` does not report any data on macro definitions but reports more accurate line numbers. Since we are not only going to analyze the code, but also going to generate the code, we used `Datrix` to locate the right line numbers. `Datrix` outputs abstract syntax graphs (ASG) in various formats including TA. We developed an analyzer to extract header-related code dependencies from the resulting ASG.

Other programming languages like C++/Java can also be analyzed. Java programs, however, use packages that are cleaner than legacy C code. There are also tools that can clean up false code dependencies in Java code. It will be interesting to know how good our partitioning algorithm can perform compare to existing Java optimizers. For C++, we believe it is necessary to do header file optimizations.

## 5.2 Related topics

The graph algorithm proposed for code partitioning is not the strongly connected component partitioning algorithm for general digraphs [1] because this is the case not all vertices in a connected component are treated the same way. Instead, we treat "define" and "use" vertices differently in order to ensure that false code dependencies are not introduced. The topological ordering is applied to generate code for a partition with a variation that cycles are removed by updating the edges incident on one vertex to be incident on other vertices in the cycle.

The code dependence partitioning is also different from the partitioning algorithms for data

Table 2: Measuring build time in seconds. Columns 2–3 compare the build time using preprocessed file with that using refactored files; columns 4–6 compare the normal build time of `gcc -c` using the original files to that using refactored files with/without the #pragma's; columns 7–9 compare the build time of `gcc -O3` optimizing options between the same sets of files. The first 15 rows are the incremental refactored files, and the total build including the other 32 files are also measured.

| Time | cpp | cpp' | gcc -c | gcc -c' | gcc -c' \# | gcc -O3 | gcc -O3' | gcc -O3'\# |
|---|---|---|---|---|---|---|---|---|
| buffer.c | 0.64 | 0.46 | 0.58 | 0.46 | 0.46 | 1.23 | 1.10 | 1.09 |
| charset.c | 0.35 | 0.16 | 0.28 | 0.16 | 0.17 | 0.50 | 0.37 | 0.36 |
| diff.c | 0.44 | 0.26 | 0.37 | 0.26 | 0.26 | 0.71 | 0.58 | 0.57 |
| digraph.c | 0.26 | 0.07 | 0.19 | 0.07 | 0.06 | 0.26 | 0.12 | 0.12 |
| edit.c | 0.72 | 0.53 | 0.65 | 0.53 | 0.54 | 1.77 | 1.64 | 1.62 |
| fold.c | 0.52 | 0.33 | 0.45 | 0.33 | 0.33 | 1.01 | 0.89 | 0.89 |
| mark.c | 0.41 | 0.23 | 0.34 | 0.23 | 0.22 | 0.81 | 0.53 | 0.53 |
| memfile.c | 0.32 | 0.14 | 0.22 | 0.14 | 0.13 | 0.34 | 0.27 | 0.27 |
| memline.c | 0.63 | 0.44 | 0.46 | 0.44 | 0.43 | 1.00 | 0.99 | 0.98 |
| misc1.c | 0.79 | 0.61 | 0.61 | 0.61 | 0.60 | 1.93 | 1.95 | 1.91 |
| ops.c | 0.75 | 0.56 | 0.62 | 0.56 | 0.56 | 1.58 | 1.61 | 1.60 |
| pty.c | 0.25 | 0.04 | 0.17 | 0.04 | 0.04 | 0.18 | 0.05 | 0.07 |
| tag.c | 0.44 | 0.26 | 0.27 | 0.26 | 0.26 | 0.51 | 0.66 | 0.66 |
| undo.c | 0.33 | 0.15 | 0.26 | 0.15 | 0.14 | 0.41 | 0.30 | 0.33 |
| version.c | 0.30 | 0.08 | 0.19 | 0.08 | 0.08 | 0.25 | 0.15 | 0.15 |
| subtotal | 7.15 | 4.32 | 5.66 | 4.32 | 4.28 | 12.49 | 11.21 | 11.15 |
| reduction | | -39.58% | | -23.67% | -32.24% | | -10.25% | -12.02% |
| other files | 19.30 | 19.30 | 13.00 | 13.00 | 13.00 | 29.77 | 29.77 | 29.77 |
| total | 26.45 | 23.62 | 18.66 | 17.32 | 17.28 | 42.26 | 40.98 | 40.92 |
| reduction | | -10.70% | | -7.18% | -7.40% | | -3.03% | -3.17% |

dependencies [13, 5]. Data dependence partitioning looks for parallelism in independent partitions, while the code dependence partitioning may create partitions that still depend on each other. The code dependence partitioning is a refactoring technique aiming at maintainability, while the data dependence partitioning is an performance optimization transformation technique. Using parallelism speeds up the execution of the code while removing code dependencies speeds up both complete and incremental builds and consequently accelerates the development process. In this sense they are complementary.

Another related topic is link-time optimization [12], which tries to remove excessive variable and function inclusions in order to make the static binary size and the dynamic code size (footprint) smaller. When the whole library headers are included, it is often safer for the developer to include the whole library to avoid link error. After refactoring the headers at the source level, such situations would be less common. A cleaner binary code can be generated if we can further refactor the libraries being linked. The combined approach may strip the unused library functions completely and thus lead to smaller binary size.

Among these possibilities, we believe the header refactoring is most suitable for speeding up the development, while time and space performance optimizations are still possible.

# 6 Future work

There are several enhancements to our work that we are currently investigating. The initial aim is to complete the refactoring of the VIM. The observations that we have made on VIM have enabled us to formulate a strategy

for tackling larger systems. In terms of potential impact, our data suggests that for a large commercial software product, there are tens of thousands of files on average. In one experiment we observed that an implementation file, on average, grows by a factor of 50. In other words, while the implementation file size is around 50 KB, the preprocessed file becomes 2.5 MB. Based on our VIM experiments and preliminary assessments, we estimate that the size of preprocessed files can be reduced, on average by 30 to 40 %. This results in a saving of around 1MB. Assuming there are 1000 files in the system, there will be 1GB of space savings. Furthermore, taking into account the continuous build paradigms used in commercial software development, this number has be multiplied by 7 to compute the weekly savings, i.e. 7 GB. Considering the average release cycle varies from 12 to 16 months, the weekly saving is multiplied by the number of weeks in the release cycle. This is a significant saving that justifies the overall effort of refactoring as put forward by this paper.

Another challenge facing a refactoring effort for a large software system involves the prevalent use of conditional compilation directives. Most parsing technologies rely on preprocessed programs and as such fail to provide any information about the conditional compilations. In other words, we do not know exactly what directive occurs on which line. Furthermore, we gather data from on possible compilation based on one set of conditional compilation directives per platform. Our initial strategy for dealing with conditional compilations involves the splitting of the conditional guards. For example, consider the header file `foo.h` shown below:

```
#ifdef FOO
    #define EXTERN extern
    EXTERN int foo ();
    EXTERN int
    bar ();
#else
    #define EXTERN
    EXTERN float moo();
    EXTERN float
    bar();
#endif
```

Here we split the header file into three files as shown below:

```
----foo.h----
#ifdef FOO
    #include "foo1.h"
#else
    #include "foo2.h"
#endif

---foo1.h----
extern int foo ();
extern int bar ();
---foo2.h----
float moo();
float bar();
```

We can proceed with refactoring as before by analyzing `foo1.h` and `foo2.h` without touching the `foo.h`. This requires enumeration of all combinations of macros used in the program as different scenarios which can be exhausting for large systems. Another way is to treat macros as code dependencies from macro units to program units. It works only when every single macro can be located accurately in the code, in which case there is only one scenario. However, such accuracy is very hard to achieve. Currently we are investigating a way to confine the scenario enumeration only to the explicit conditional directives, while treating other macros the same way as other program units. In this way we hope to get a fast and accurate solution in the near future.

# 7 Conclusions

False dependencies among implementation files in large software systems can significantly increase the build time. This paper described a refactoring tool for discovering and removing these dependencies from the header inclusions. This tool can be applied incrementally to a subset of files or its scope can be adjusted include one or more types of program units (e.g. functions, global variables to type definitions). As shown through a series of experiments, the resulting code is substantially reduced, thereby reducing build time.

# About the Authors

Dr. Yijun Yu is a research associate at the University of Toronto. His Internet address is http://www.cs.toronto.edu/∼yijun.

Dr. Homy Dayani-Fard is a member of the IBM Toronto Lab. His Internet address is http://www.cs.queensu.ca/∼dayani.

Prof. John Mylopoulos is the director of the Knowledge Management lab at the University of Toronto. His Internet address is http://www.cs.toronto.edu/∼jm.

# References

[1] Jorgen Bang-Jensen and Gregory Gutin. *Digraph: Theory, Algorithms and Applications*. Springer-Verlag, 2002.

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

[3] M. A. Cusumano and R. W. Selby. How Microsoft builds software. *Communications of the ACM*, 40(6), June 1997.

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

[5] Erik H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, jul 1992.

[6] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, JUL 1987.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Working Conference on Reverse Engineering*, October 1998.

[9] Richard C. Holt, Ahmed E. Hassan, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R schema for the Datrix C/C++/Java exchange format. In *Working Conference on Reverse Engineering*, pages 284–286, 2000.

[10] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE '00*, November 2000.

[11] Bram Moolenaar. Vim 6.1, http://www.vim.org, 2003.

[12] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. Alto: A link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 1 2001.

[13] Y. Yu and E. D'Hollander. Partitioning loops with variable dependence distances. In *Proceedings of 2000 International Conference on Parallel Processing (29th ICPP'00)*, Toronto, Canada, August 2000. Ohio State Univ.