ISSN 1744-1986

Technical Report N° 2012/ 07

Faster Compilation through Lighter Precompilation

Yijun Yu

, 2012

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

Faster Compilation through Lighter Precompilation

Yijun Yu
Department of Computing, The Open University

December 18, 2012

Abstract

Existing C/C++ precompilers [9, 2] remove false dependencies between C/C++ headers to speed up their incremental compilation. Whilst parsing costs less than the backend optimisations in modern compilers, the overhead of restructuring exceeds the gain from a faster parsing. This report summaries the steps to reduce precompilation overhead through a lighter tag analyser. It also presents an evaluation on the tool.

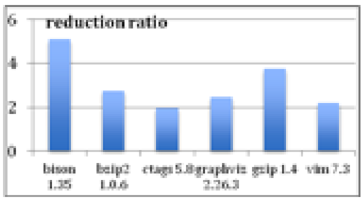
Introduction

Current state-of-the-art for faster C/C++ compilation employs a farm of compiling servers for parallel builds [8], and makes use of caching [1] and precompiled header techniques [3, 4, 7] to avoid repeated parsing. Branching from the development of gcc, Apple Incs clang compiler has an extension to use #include statement removal technique to speedup the compilation for Mac OSX based development [2].

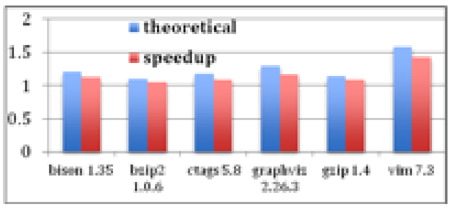
In an earlier academic paper [10], we demonstrated the algorithm that reduces the headers size at the precompilation step, in-between the preprocessing step and the parsing step in normal compilations. The steps in compilation are illustrated by the conceptual process diagram of gcc. Comparing to the approach that only removes extraneous #include statements [2], our improvement is optimal in the sense that it removes false dependencies (i.e., defines-uses) among program entities inside the header/compilation units [9]. For example, an inclusion of <stdio.h> could introduce 843 LOC to the precompiled image, whilst a Hello World program only requires the prototype of printf function in one line of code.

Cleaning up the APIs for unused program entities can greatly reduce the compilation time of incremental builds: applied to the IBM DB2 product, the technique reduced the time of incremental compilations of 14 million LOC from 7 hours to 2 hours [10]. When combined with other state-of-the-art compilation optimization techniques, a super linear speedup was recorded, e.g. 40 times out of a cluster of 8 cores. The correctness of precompilation is guaranteed because the resulting binary is compatible to the normal compilation of the original code. This process is also made transparent to the development flow by keeping #line directives in the resulting code so that they can be traced by debugging and testing tasks.

Limitations of previous work: Precompiled header techniques are adopted in modern C/C++ compilers [3, 4, 7, 6]. They work by paying the overhead of storing symbol tables of the headers at the first time of parsing, in exchange for speed gains while loading them for subsequent compilations. Common to these techniques, including our earlier work [10], fresh compilation for the first-time slows down typically by 2–3 times due to the overhead of the precompilation step added to the compilation



(a) size reduction



(b) fresh build speedup

Figure 1: Current performance on C/C++ benchmarks

process. Therefore, instead of reusing a C/C++ parser to perform precompilation, one needs a lighter approach. Furthermore, existing API extraction work do not further analyze whether they can be made thinner and faster for both build-time and runtime integrations. Little has been done on checking how the APIs are consistent or compliant to the program naming conventions.

Our Approach

To address the limitations of precompilation for fresh builds, we have adapted the Exuberant ctags tool for improving the fresh builds, that is, gaining speed already at the very first time of compilation of C/C++ programs. Exuberant ctags [5] is an open-source implementation of the well-known ctags program, which creates tags of program entities to assist programmers navigating between them inside text editors such as vim and emacs. Although the tool is not designed for the task of precompilation, we have modified it to check dependencies on previous entities in the lexical order for opportunities of false dependency removal.

Evaluation

As a result, thinner header interfaces are obtained, consuming less time than the gain in a faster compilation of the reduced code.

In the experimental evaluation, we used a benchmark of six open-source programs. Table 1 lists the lines of code (LOC) metrics of the programs, in terms of both the

Table 1: The size of the benchmarks

program	original LOC	restructured LOC
bison 1.35	38723	7581
bzip2 1.0.6	15800	5727
ctags 5.8	91096	45889
graphviz 2.26.3	924845	379096
gzip 1.4	17992	4773
vim 7.3	473258	217251

Table 2: The speedup ratios of the benchmarks

program	theoretical speedup	practical speedup
bison 1.35	1.21	1.13
bzip2 1.0.6	1.10	1.05
ctags 5.8	1.17	1.09
graphviz 2.26.3	1.29	1.17
gzip 1.4	1.13	1.09
vim 7.3	1.58	1.43

original size and the reduced size.

Table 2 lists the speedup in terms of both theoretical (assuming no restructuring overhead) and practical ratios (taking into account the restructuring time).

Figure lists a few characteristics of such improvement for the fresh compilation. First of all, header size reduction is substantial, up to 2 5 times smaller than the original preprocessed images. Both the computation of program entity dependencies and the reductions are achieved within the time of original compilation. In other words, the overhead is fully compensated by the time gained in fresh compilation of smaller footprints. Without such an overhead, the ideal fresh build speedup is e.g. around 1.58 times faster for the vim case. With the overhead of the current implementation of lighter precompilation, the speedup is e.g. around 1.43 times faster for the vim case. There is still more room to tune the performance of the ctags-based precompilation, e.g., by using a more efficient string hash function and by using a Map-Reduce algorithm to exploit the explicit parallelism in tag processing.

Conclusion

As with the earlier experiment in [10], we found that the reduced code can improve the performance of all existing C/C++ compilers [3, 4, 7, 6].

Bibliography

[1] Andrew Tridgell and Joel Rosdahl. ccache a fast C/C++ compiler cache.

[2] Craig Silverstein, Chandler Carruth, Kaelyn Uhrain, Paul Holden, Fabio Fracassi, and Volodymyr Sapsai. A tool for use with clang to analyze #includes in c and c++ source files, http://code.google.com/p/include-what-you-use/.

[3] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[4] Intel. Intel c compiler, December 2012. Page Version ID: 527410952.

[5] Ken Arnold. Exuberant ctags, December 2012. Page Version ID: 506967980.

[6] Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[7] Microsoft. Visual c, December 2012. Page Version ID: 523585508.

[8] Martin Pool. distcc: a fast, free distributed C/C++ compiler. Technical report, Samba, 2004.

[9] Yijun Yu, Homayoun Dayani-Fard, and John Mylopoulos. Removing false code dependencies to speedup software build processes. In *CASCON*, pages 343–352, 2003.

[10] Yijun Yu, Homayoun Dayani-Fard, John Mylopoulos, and Periklis Andritsos. Reducing build time through precompilations for evolving large software. In *ICSM*, pages 59–68, 2005.