Accelerating Regression Test of Compilers by Test Program Merging

Takayuki Fukumoto [1] Kazushi Morimoto [2] Nagisa Ishiura [1]

[1] School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan
[2] Securities Systems Management Services Department, Nomura Research Institute, Tokyo, Japan

Abstract—This paper proposes a method of accelerating regression test of compilers by merging test programs in compiler test suites. Large amount of computation time is needed for compiler testing through test suites, for they consist of a huge number of test programs. Especially, in early stages of compiler development, reduction of time for testing is a critical issue, for bug fixes and regression tests are alternately repeated for many times. The proposed method attempts to shorten the time for test suite run by merging test programs in the test suite into longer but fewer programs, which drastically reduces the overhead for file open/close. During the merger, conflicts among the names of global variables, functions, and user defined types are avoided by prefixing. Header file inclusion as well as multiplier compilation are carefully handled so that the semantics of the original test programs are maintained. A technique is also proposed to identify test programs that resulted in execution errors while executing the merged test programs. In an experiment where about 9,000 test programs in the *testgen2* test suite were merged into 117 programs, computation time was reduced into 1/11.1 on Ubuntu Linux and into 1/63.9 on Cygwin on 2.5GHz Core i5 CPU.

I. Introduction

Compiler reliability is a critical issue in all kinds of system development, for compilers are infrastructure tools to develop various software. Compilers must be tested thoroughly in many ways which include testing by test suites [1, 2, 3, 4, 5], random testing [6, 7, 8, 9], etc.

A test suite of compilers is a set of test programs which are designed to check compilers' functionalities. The size of the test suites tend to be huge, for they must test compilers with respect to a broad range of requirements from the language specifications. The number of test programs in a test suite may range from several thousands to millions, which results in very long run time.

The run time might not be a big issue when the test suite is run only once. However, as long as bugs are detected, they must be fixed, after which the test must be rerun in order to verify that the bugs has been removed and also that new bugs are not introduced by the modification. Especially in early stages of compiler development, where test suites still detects a lot of errors, bug fixes and regression tests are frequently repeated, where the number of remaining bugs reported by the test suite works as a barometer of the progress of the debugging tasks. In this situation, reduction of the time for test is of critical importance.

This paper proposes a method of reducing the time for test suite run drastically by *merging* test programs. A group of test programs are merged into a single program where the main functions in the original programs are converted into subroutine functions that are called from a new main function. Conflicts of the names are resolved by renaming and file inclusion and multifile compilation are carefully handled so as to prevent the merging from changing the intent of the original test programs. Furthermore, a technique is also proposed to identify test programs that resulted in execution errors while executing the merged test programs.

Experiments on the *testgen2* C compiler test suite [5] shows that the run time was reduced into 1/11.1 and 1/63.9 of those for the original test suite on Ubuntu Linux and Cygwin, respectively.

II. Run Time for Test Suites for C Compilers

One of the most popular way of testing compilers is to compile and execute a set of programs to see if the generated codes behave as expected. The programs are called test programs and the set of the test programs are called a test suite.

A compiler test suite usually consists of a huge number of test programs. A primary reason for this is that compilers have so many functionalities to be validated. Another reason is that the test programs are often broken into smaller pieces so that the debugging of the compilers will be easier when errors are detected. This makes the run time for test suites longer for its LOCs, due to increased overhead for file open/close.

There are several types of test suites for C compilers. Validation test suites [1] and commercial test suites [2, 3], which are used for mature compilers, consist of hundred thousands to millions of test programs. The test suite distributed with GCC (GNU Compiler Collection) [4] is

```
01: #ifdef SYSDEP_H
02: #include "sysdep.h"
03: #endif
04: #include "stnd.h"
05:
06: main(){
07:    static int Variable;
08:
09:    itest = 0;
10:    Variable = 1;
11:    itest = Variable;
12:
13:    if (itest == 1)
14:       printok();
15:    else
16:       printno();
17:
18:    return 0;
19: }
```

Fig. 1. An example of test programs.

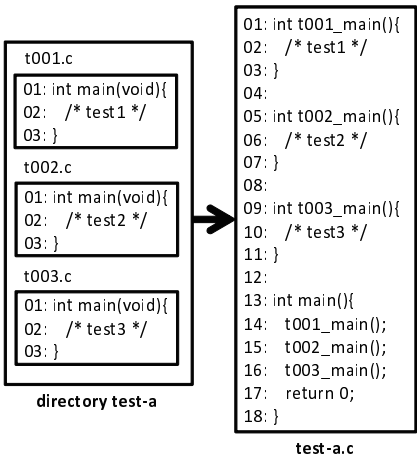


Fig. 2. Basic concept of test program merging.

composed of abut 3,000 files and is also used to verify the correctness of retargeting.

On the other hand, some test suites, such as *testgen2* [5], are intended for use during compiler development. The *testgen2* test suite is a collection of about 9,000 test programs to check basic functionalities of C compilers. Fig. 1 is an example of the test programs in *testgen2*. It checks if accesses to a static variable work correctly, where `printok()` and `printno()` are user definable macros to report the result[1].

During compiler development, the test suite should be run every time bugs are fixed, for it must be confirmed that the bugs have been certainly removed and that the functionalities which had been valid before the bug fix have not been impaired. On 2.5 GHz Core i5 CPU with 4GB memory, the run time for the *testgen2* test suite is about 7 minutes on Ubuntu Linux and 4 hours 49 minutes on Cygwin, while the time necessary for building GCC (version 4.5.1) is about 11.5 minutes on Ubuntu Linux and 39.5 minutes on Cygwin. In this situation, acceleration of test suite run becomes a very important issue, especially on Cygwin where file accesses are slow.

III. Acceleration of Test Suite Run by Test Program Merging

A. Basic Idea

This paper proposes a program merging technique for accelerating test suite run, where small test programs are combined into larger but fewer programs so that the overhead for the file accesses will be reduced.

Fig. 2 illustrates the basic concept of the proposed method. There are three test programs `t001.c`, `t002.c`, and `t003.c` in a directory `test-a`. They are merged into a single program `test-a.c`, where main functions in the original test programs are renamed into `t001_main`, `t002_main`, and `t003_main`, and are invoked from the new main function of `test-a.c`. The resulting program is functionally equivalent to the three original test programs, while it is more efficient in terms of the file I/O (open/close) count. Overhead for launching the compiler is also reduced.

Although the basic merger scheme is simple, conflicts of the variable, function and type names must be taken care of. Special treatments are also needed for file inclusion and multifile compilation.

B. Resolving Name Conflicts by Prefixing

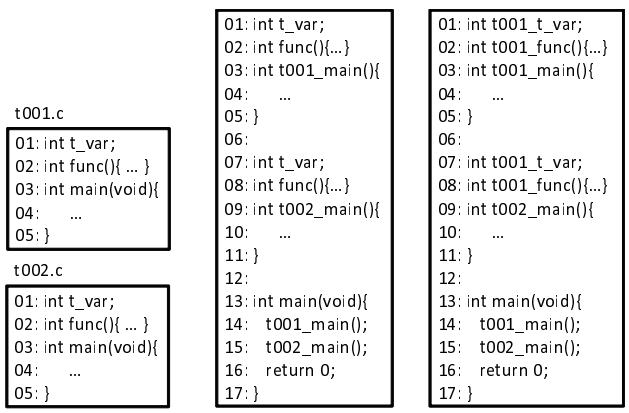
Naive merger illustrated in Fig. 2 may cause conflicts of the names. Cares must be taken on the names of the global variables, the functions, and the types (declared by `typedef` statements). Fig. 3 illustrates an example of the conflicts. Test programs `t001.c` and `t002.c` both declare global variable `t_var` and function `func`. Fig. 3 (b) is a result of naive merger where conflicts occur with respect to names `t_var` and `func`.

This kind of conflicts are resolved by prefixing all the names of the variables, functions, and types in the test programs by strings identifying the original programs. Fig. 3 (c) shows the example where distinct names are given to the global variables and functions.

C. Handling of Header Files

Header files are classified into two categories; the system (or standard) header files and user defined header files. They must be handled differently according to their categories.

[1]Usually `printf()` is used to output specific strings to the standard output, but in early development stages where `printf()` is not yet available, low level I/O functions like `write()` or direct access to the memory is used instead.



(a) before merger (b) naive merger (c) prefixing

Fig. 3. Resolution of name conflicts.

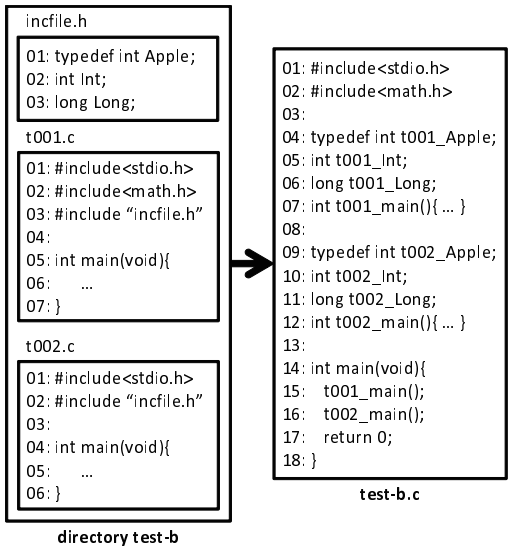


Fig. 4. Handling of header files.

The system header files are common to all the programs. Since all the functions, macro variables, and global variables defined in the system header files are referenced from the other functions by the same name, their names need not be be modified. Although almost all the system header files are written so that they can be included multiple times, the same system header files should be included only once for safety.

On the other hand, the user header files involve the aliasing problem; different header files may define different variables or functions in the same name, which should be renamed as stated in the previous subsection.

Based on the observation above, header files are handled in the following way.

1. The system header files are not expanded. If there are multiple occurrences of inclusion for the same header file, they are reduced to one.

2. The user header files are expanded into the test programs before they are merged. During the merger, the names of all the variables, functions, and types are modified as described in the previous section.

Fig. 4 shows an example of header file processing. Directory `test-b` contains two test programs `t001.c` and `t002.c`, where `t001.c` includes standard header files `stdio.h` and `math.h` along with user header file `incfile.h`, while `t002.c` includes `stdio.h` and `incfile.h`. The program `test-b.c` on the right hand side is the result of the merger. The two system header files are included only once (on lines 01 and 02). Lines 04 through 06 are the expansion of user header file `inclfile.h`, which is the result from the include statement on line 03 of `t001.c`. The names are modified to have prefix "`t001_`." Lines 09 through 11 are also the expansion of `inclfile.h` but "`t002_`" are prefixed to the names this time.

The procedure above is easily implemented if C preprocessors can selectively handle the system and user header files. For example, gcc with `-E -nostdinc` option expands only user header files but ignores include statements for the system header files.

D. *Merging of Input/Output Data*

If test programs are accompanied with input data and expected output data, they must also be merged. This is rather straightforward; they just need to be concatenated. Sometimes, as is in the *testgen2* test suite, the occurrences of specific strings are counted. In that case, the sum of the expected counts are used for the merged test programs.

E. *Multifile Compilation*

Test suites may contain tests to validate multifile compilation. In this case, the simple prefixing scheme described in subsection III.B. does not work, for the same program file is used multiple times in different combinations of other files. To solve this problem, the renaming scheme is extended so that the prefixes will be the concatenations of the identifying strings of the program files used for multifile compilation.

Fig. 5 illustrates this scheme. Directory `test-c` has four program files where `A.c` contains a main function. A test scenario for multifile compilation is assumed to be specified in file `FILESET`. In this example, two test cases, linking of files {`A.c`, `B.c`, `C.c`} and linking of {`A.c`, `D.c`}, are instructed. The test programs in directory `test-c` are merged into three files `test-c1.c`, `test-c2.c`, `test-c3.c`, and a new `FILESET` file. The file `test-`i`.c` contains the programs that appears as the i-th elements

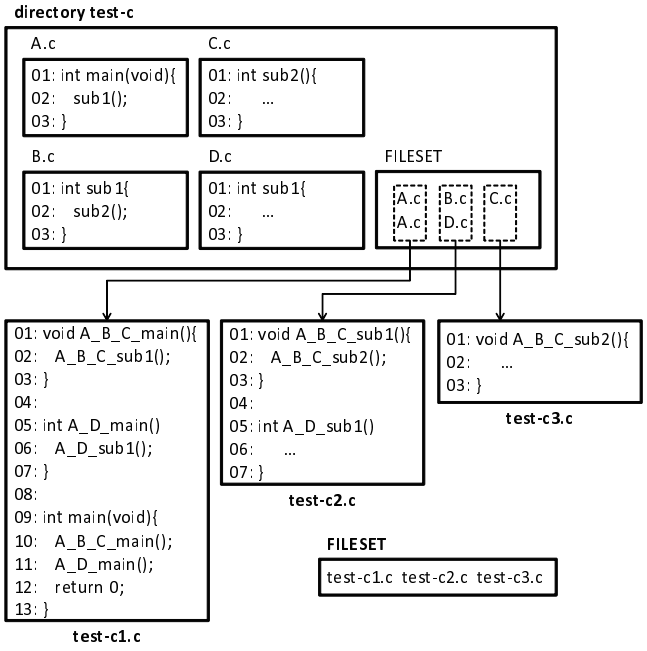


Fig. 5. Merger of multifile compilation tests.

in the combinations specified in the original `FILESET`. `test-1.c` also contains the new main function to call all the test cases. Note that all the names of the variables, functions, and types are prefixed with the combination of the name of the files that are tested for multifile compilation. For example, all the variables and functions used for test of combination {`A.c`, `B.c`,`C.c`} are renamed to have prefix "`A_B_C_`."

IV. Identification of Error Programs

The method described so far is only effective in confirming that the compiler under test passes all the tests. Once an error is detected by a merged test program, the original tests should be run to identify the test cases in question. In the actual compiler development, the test suite run is repeated for many times even when the compiler fails at many test cases, for the number of remaining (faulty) cases works as a barometer of the progress of compiler development or debugging.

In this section, the merger scheme is extended so that the test programs at which the compiler under test fails are identified by the run of the merged test program. This is realized by letting the merged programs output special delimiters to separate the outputs of the original test programs and also by making the merged program resumable even if some test cases result in crashes.

Fig. 6 shows how the delimiter works. Suppose test two programs are merged into one. Fig. 6 (a) shows the two output sequences of the original programs where `@NG@` in

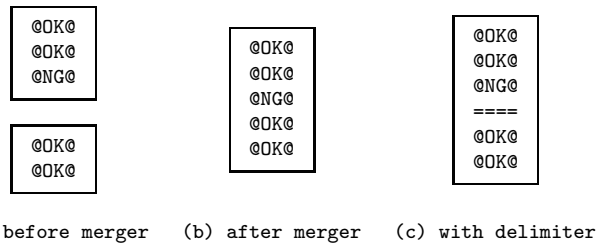


Fig. 6. Output of the test programs.

the first sequence indicates that the compiler has some problem with the first test case. However, the test programs are merged, the output becomes as shown in Fig. 6 (b), from which one cannot identify the test program that caused the error. This problem is solved by letting the merged test program output the delimiter to separate the outputs of the test case, as in Fig. 6 (c).

The other problem to be considered is unexpected abortion of test caused by crash or timeout (to break infinite looping) of the generated codes or of the compiler itself. If there is a test program that crashes in a group of programs to be merged, the resulting program also crashes, leaving the test cases after the crash unexecuted. In order to precisely identify the programs that does not pass the test, execution must be resumed after the abortion.

This is realized by extending both merged test programs and the test script. On finish or abortion of the run of the merged test program, the test script compares the output of the test program with the expected values to check if all the test cases have been executed. If abortion is detected, the test case in question is identified and the test program is restarted. The merged test program is modified so that it can resume execution from the test case specified by one of the command line arguments. Fig. 7 is a modified version of the merged program. It can choose the entry point according to the argument passed from the command line. `PRINTDIV()` on lines 12 and 14 is a macro to output the delimiter.

V. Experimental Results

A test merger script and a test execution script based on the proposed method were implemented on top of the *testgen2* test suite. The scripts were written in Perl 5.10.1 and runs on Unix systems such as Linux, Cygwin, and Mac OS X.

The merger was applied to the K&R directories of the *testgen2* test suite. About 9,000 program files in 85 directories were merged into 117 files. In the current implementation, the test programs in a directory were merged into one test program. The reason why there were more than 85 files was that some directories included tests for

```
01: int t001_main(){
02:   /* test1 */
03: }
04:
05: int t002_main(){
06:   /* test2 */
07: }
08:
09: int main(int ac, int *av[])
10:   switch(INT(av[1])){
11:     case1: t001_main();
12:             PRINTDIV();
13:     case2: t002_main();
14:             PRINTDIV();
15:   }
16:   return 0;
17: }
```

Fig. 7. Revised version of merged test program.

multifile compilation. [2].

TABLEs I and II summarize the run time for the *testgen2* test suite on Ubuntu Linux (GCC 4.5.2) and on Cygwin (GCC 3.4.4), respectively. The CPU was 2.5 GHz Core i5 and was equipped with 4.0GB memory. In column "settings," "x% fail" means that the tests fail at x% of the programs in the test suite (this situation was created by modifying the expected output for x% of the test cases) and "x% abortion" means that the execution of was aborted at x% of the test programs (this situation was created by intentionally injecting a code to cause memory access error into x% of the test programs). Columns "w/o merge" and "proposed" show run time for the test suite before merger and after merger, respectively.

When the compilers passes all the tests, the test suite run was accelerated by 11.1 times on Linux and 63.9 times on Cygwin. Especially on Cygwin, where file accesses are slow, reduction of run time is significant. As rows "20% fail" and "40% fail" show, the performance of the merged test programs was not effected by the percentage of test cases that fails. However, frequent abortion of test programs degrades the performance of the merged test programs. However, even when 40% of the tests result in abortion, which might be rather too pessimistic, the merged program was still faster by more than 5 times.

The time required for merger was 2 minutes 30 seconds on Linux Linux and 2 hours 40 minutes on Cygwin. Note that the merge have to be done only once, as long as there is no change on the test suite.

Although the merged programs theoretically have the same functionality as the original test programs, they might behave differently depending potential compiler malfunctions. In fact, there were cases where GCC 4.5.1 cross compiler for ARM passes all the original test programs in the test suite but it fails on a merged program. The cause of the error was not indentified yet, but our conjecture is that merged programs have higher testing ability than the original test programs. On the other hand, no opposite case has been encountered so far where a merged test program failed to detect errors in the original programs.

[2]More specifically, there are 9 directories out of 85 that include testing of multifile compilation. Linking of 2 files are required in 6 directories, and linking of 4, 10, and 15 files are required in one directory each. Thus all the test cases are merged into $(85-9)+2\times 6+4+10+15=117$ files

TABLE I
RUN TIME OF *testgen2* TEST SUITE ON UBUNTU LINUX.

settings	w/o merge	proposed	acceleration
0% fail	7m03s	38s	11.1
20% fail	7m03s	39s	10.8
40% fail	7m03s	38s	11.1
20% abortion	6m54s	56s	7.4
40% abortion	6m49s	1m19s	5.2

GCC 4.5.2 (i686-linux-gnu)/Core i5 2.5GHz with 4GB memory

TABLE II
RUN TIME OF *testgen2* TEST SUITE ON CYGWIN.

settings	w/o merge	proposed	acceleration
0% fail	4h49m48s	4m32s	63.9
20% fail	4h49m03s	4m35s	63.1
40% fail	4h48m38s	4m32s	63.7
20% abortion	4h47m52s	24m28s	11.8
40% abortion	4h49m11s	45m16s	6.4

GCC 3.4.4 (i686-pc-cygwin)/Core i5 2.5GHz with 4GB memory

VI. CONCLUSION

A method of accelerating regression test of compilers by merging programs in test suite was presented in this paper, by which computation time for test suite run was drastically reduced. The scripts based on the proposed method will be distributed under GPL2 along with the *testgen2* test suite. As another approach to accelerating test suite run, we are now developing multicore versions of the test script.

ACKNOWLEDGMENT

Authors would like to thank Mr. Nobuyuki Hikichi (TOPS Systems Corporation) for his discussion and advices on compiler development and test suite. We would also like to thank Mr. Shohei Yoshida (AXE, Inc.) and Mr. Soichiro Taga (now with Mitsubishi Electric Micro-Computer Application Software Co., Ltd.) for his comments and continuous support for improving *testgen2*test suite. We thank Mr. Ooki and all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion on this research.

References

[1] http://www.plumhall.com/suites.html.

[2] http://www.actest.co.uk/.

[3] http://www.ace.nl/compiler/supertest.html.

[4] http://gcc.gnu.org/install/test.html.

[5] http://ist.ksc.kwansei.ac.jp/ ishiura/pub/testgen2/index.html.

[6] Christian Lindig: "Find a Compiler Bug in 5 Minutes," in *Proc. ACM International Symposium on Automated Analysis-Driven Debugging*, pp. 3–12 (Sept. 2005).

[7] Eric Eide and John Regehr: "Volatiles Are Miscompiled, and What to Do about It," in *Proc. 7th ACM International Conference on Embedded Software*, pp. 255–264 (Oct. 2008).

[8] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr: "Finding and Understanding Bugs in C Compilers," in *Proc. 2011 ACM SIGPLAN Conference on Programing Language Design and Implementation (PLDI)* pp. 283–294 (June 2011).

[9] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda: "Random Testing of C Compilers Targeting Arithmetic Optimization," in *Proc. SASIMI 2012*, R1-10 (Mar. 2012).