# University of Houston

# Program Analysis and Testing

# GCC Project

GitHub URL:
https://github.com/dhondiapoorva/gcc_analysis.git


Team:

Viswanath, Puppala (2084700)

Apoorva Dhondi (2148722)

Nitish Nannapaneni (2089006)

# GCC – TEST SUITE ANALYSIS

## Introduction

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for Linux, Windows, various BSDs, and a wide assortment of other operating systems, also for several major programming languages. These languages currently include C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada. In this project we are going to build and run the test suite designed of the GCC developers and analyze the test suite metrics and performance of it.

## Assumptions

- We have tried multiple releases of the project and different OS versions of Ubuntu. But only branch source code which we were successful to build and run the test suite was 'release/gcc-11' and in Ubuntu 20.04 LTS
- The test suite of this project is huge, and it will take hours or days to complete the execution. Hence all the figures and information displayed in this report are after running a subset of the test suite. However, we have included all the steps to completely run the test suite and the gen_figures.py will generate data accordingly

## Building

To build the project, we need to download the repository from here.

1. Create a new directory for the object files parallel to the git repository you have downloaded.
2. Download the prerequisites by running this command in gcc folder
    a. ./contrib/download_prerequisites
    b. You may need to install flex for processing the .lex files (sudo apt install flex)
    c. mkdir objdir; cd objdir
3. In the objdir, run the following command
    a. ../gcc/configure –enable-coverage –disable-multilib –disable-werror
        i. –enable-coverage: This option will generate the .gcda and .gcno files which are required to generate the coverage reports
        ii. –disable-multiib: since my machine is 64 bit I don't want 32 bit files when building
        iii. –disable-werror: This will enable the builder to run by skipping the warnings
4. To run build the code run inside the objdir
    a. make -j 4
        i. -j is for compiling the code parallelly on multiple cores
5. You need to install DejaGnu for running the test suite
6. After this you are ready to run the test suite. To run the complete test suite (which may take hours or days), run
    a. make -j n -k check

7. To generate html reports, install gcovr (pip install gcovr). Inside the gcc of objdir run
    a. gcovr --gcov-ignore-parse-errors -r ../../../ --html -o coverage.html
8. To generate CSV reports. CSV reports are more informative than html reports
    a. gcovr --gcov-ignore-parse-errors -r ../../../ --csv -o coverage.csv

Rest of the figures and information is generated by the python script.

## Analysis

### Folder Structure

Since GCC is the core compiler for many languages (c, c++, ada, Fortran etc) the source code and test suite are humongous. Overall, there are 79559 test files written for all the languages in the test suite folder. Each language has unit test cases as well as compiler specific test cases. Whenever the test suite is triggered a change.log file will be generated in the directory which stores the console logs of that run.

```
viswa@LAPTOP-KCFNBVCI:~/gcc/gcc/testsuite$ find . -maxdepth 1 -type d | while read -r dir
> do printf "%s:\t" "$dir"; find "$dir" -type f | wc -l; done
.:      79559
./ada:  2604
./selftests:    21
./g++.target:   298
./gfortran.dg:  7076
./go.test:      1942
./jit.dg:       135
./gcc.misc-tests:       72
./g++.old-deja: 3237
./go.go-torture:        33
./lib:  67
./brig.dg:      16
./gcc.target:   22452
./gdc.dg:       246
./config:       1
./gfortran.fortran-torture:     328
./gdc.test:     2404
./obj-c++.dg:   386
./go.dg:        9
./gcc.c-torture:        3706
./gcc.src:      1
./objc-obj-c++-shared:  46
./objc.dg:      454
./gcc.dg:       14224
./gcc.test-framework:   59
./objc: 111
./gcc.dg-selftests:     1
./g++.dg:       15287
./gnat.dg:      2458
./c-c++-common: 1862
```

### Test Suite Execution

As previously said, the test suite is extensive, but its visibility and feasibility are properly maintained. The logs are conveniently saved in the appropriate directories, allowing us to debug when we encounter problems. The developers have been gracious in allowing make options to run in many ways. We can run language-specific test cases such as make check-c,

make check-fortran, and so on, or we can run individual test files. This allows us to run the test cases for a certain module whenever we need them. As a result, we have more control and versatility over the test outfit. Depending on the system configuration and parallelism of the execution, the time it takes to perform the entire test suite might range from 20 hours to days. The modular test execution of the test suite is useful for running test cases for certain components while avoiding the lengthy execution times of the entire test suite.

By designing the test cases what are you eating every component of a module, the overall quality of the test suite and test cases is maintained. The huge number of test cases suggests that new test cases are added to the suite on a regular basis. Because this project has been maintained for more than two decades, it has a huge number of test cases.

When executing a test cases, with the –enable-coverage option the DejaGnu framework generates the .gcda and .gcno files which are used by gcov to keep track of the coverage reports. Generating the coverage reports is bit of a headache. There are a lot of compatibility issues with the coverage files and gcov. To generate the visual representation of the reports we need either to use gcovr or lcov.

## State of Testing

The State of Testing has involved many advancements and improvisations through out the project. Since 1997 to 2006, the adequacy of tests was much lesser, it has been gradually increasing from 2007 to present. Upon our analysis, the following are the results obtained for the number of test files present to the total of number of assert statements or debug statements in testsuite and in the production files.

```
$ cat overall_analysis.txt
Total Number of Test Files = 84878
Total Number of Assert Statements in Test Files = 446
```

```
$ cat overall_prod_analysis.txt
Total Number of Prod Files = 115730
Total Number of Assert Statements in Prod Files = 67814
Total Number of Debug Statements in Prod Files = 8869
```

## Developer vs Tester

From the perspective of a tester, this test suite is beneficial since the high number of test cases will ensure that past functionality is not disrupted by new advancements. New unit test cases for the newly introduced feature are immediately added to the test suite. They used fuzzer to create test cases for a couple of the components, such as C and Ada. Given the frequency with which test cases are added, the tester would have a lot of control and visibility over the faults and coverage reports. The tester may easily track the development and bug fixes this way.

Adding test cases exponentially for each development and running the lengthy test suite is a time-consuming operation from the developer's perspective. To run the specific test suite on his or her module, the developer would have to wait at least hours. This would cause the development process to be slowed down.

However, given that the project's source code contains thousands of lines of code, it's fair to expect thousands of test cases and a huge test suite. When run on 7 cores, the execution times of each module may take a few of hours on average. The test suite has a code coverage of nearly 4.4 percent (according to generated html. We did not run the complete test suite).

## Adequacy of Tests

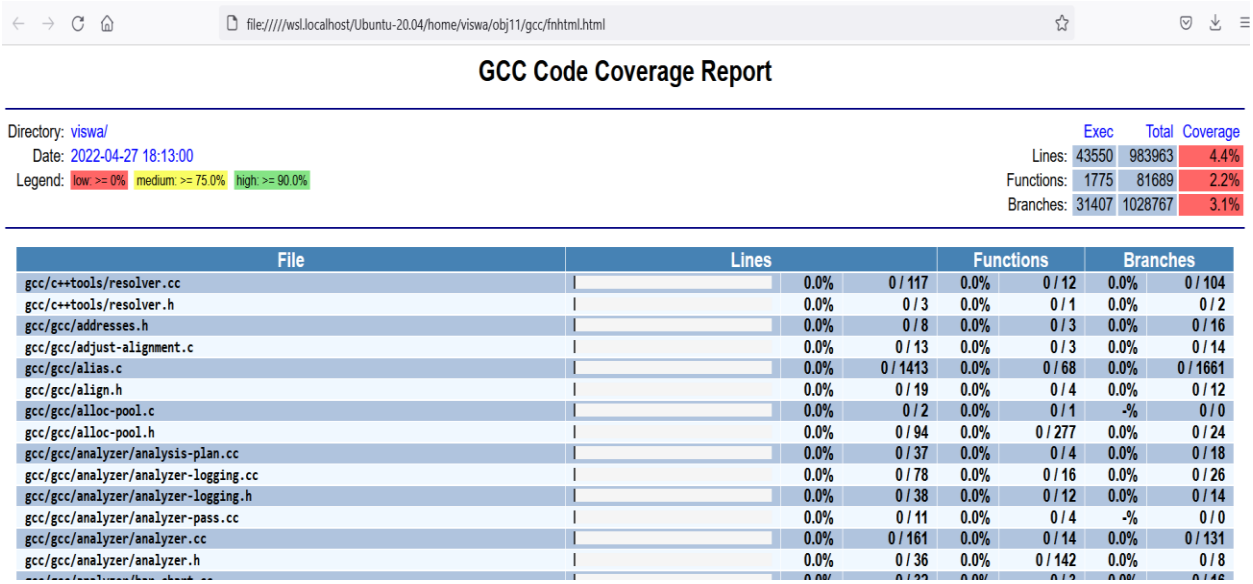
A test adequacy criterion is a predicate that is true (satisfied) or false (not satisfied) of a program, test suite  pair. The adequacy criterion is then satisfied if every test obligation is satisfied by at least one test case in the suite. We say a test suite satisfies an adequacy criterion if all the tests succeed and if every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.

For example, the statement coverage adequacy criterion is satisfied by a particular test suite for a particular program if each executable statement in the program is executed by at least one test case in the test suite.

In this project, there are about  84,878 test files of which only 446 assert statements have been covered under all the test cases, which describes the inadequacy of the test cases and does not satisfy the criterion. The following information gives detailed explanation of the code coverage.

## Code coverage

We have generated the code coverage reports using gcovr (pip install gcovr) which provides the line, function and code coverage reports for the source files. This must be run in the objdir/gcc folder. We have run only test suite for ada, fortran and c language. This is a snippet of the coverage generated but when we run the *make -k check* it generates the report for all the files and it will have a greater coverage metrics. However as mentioned in the above, from a tester standpoint running the complete test suite to generate the coverage reports before every release will be hard to maintain and consumes time. The function coverage of the source code is 2.2%. The branch coverage is around 3.1% and the code coverage is 4.4%.



file:////wsl.localhost/Ubuntu-20.04/home/viswa/obj11/gcc/fnhtml.html

**GCC Code Coverage Report**

Directory: viswa/
Date: 2022-04-27 18:13:00
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%

| | Exec | Total | Coverage |
|---|---|---|---|
| Lines: | 43550 | 983963 | 4.4% |
| Functions: | 1775 | 81689 | 2.2% |
| Branches: | 31407 | 1028767 | 3.1% |

| File | Lines | | | Functions | | Branches | |
|---|---|---|---|---|---|---|---|
| gcc/c++tools/resolver.cc | | 0.0% | 0 / 117 | 0.0% | 0 / 12 | 0.0% | 0 / 104 |
| gcc/c++tools/resolver.h | | 0.0% | 0 / 3 | 0.0% | 0 / 1 | 0.0% | 0 / 2 |
| gcc/gcc/addresses.h | | 0.0% | 0 / 8 | 0.0% | 0 / 3 | 0.0% | 0 / 16 |
| gcc/gcc/adjust-alignment.c | | 0.0% | 0 / 13 | 0.0% | 0 / 3 | 0.0% | 0 / 14 |
| gcc/gcc/alias.c | | 0.0% | 0 / 1413 | 0.0% | 0 / 68 | 0.0% | 0 / 1661 |
| gcc/gcc/align.h | | 0.0% | 0 / 19 | 0.0% | 0 / 4 | 0.0% | 0 / 12 |
| gcc/gcc/alloc-pool.c | | 0.0% | 0 / 2 | 0.0% | 0 / 1 | -% | 0 / 0 |
| gcc/gcc/alloc-pool.h | | 0.0% | 0 / 94 | 0.0% | 0 / 277 | 0.0% | 0 / 24 |
| gcc/gcc/analyzer/analysis-plan.cc | | 0.0% | 0 / 37 | 0.0% | 0 / 4 | 0.0% | 0 / 18 |
| gcc/gcc/analyzer/analyzer-logging.cc | | 0.0% | 0 / 78 | 0.0% | 0 / 16 | 0.0% | 0 / 26 |
| gcc/gcc/analyzer/analyzer-logging.h | | 0.0% | 0 / 38 | 0.0% | 0 / 12 | 0.0% | 0 / 14 |
| gcc/gcc/analyzer/analyzer-pass.cc | | 0.0% | 0 / 11 | 0.0% | 0 / 4 | -% | 0 / 0 |
| gcc/gcc/analyzer/analyzer.cc | | 0.0% | 0 / 161 | 0.0% | 0 / 14 | 0.0% | 0 / 131 |
| gcc/gcc/analyzer/analyzer.h | | 0.0% | 0 / 36 | 0.0% | 0 / 142 | 0.0% | 0 / 8 |

## Assert Statement in Test Files

The assert statements are used to figure out how the test cases are progressing. However, the ratio of assert statements to files is less than 0.5 percent. It was difficult for me to grasp why a certain test scenario was selected. It's critical to keep the test runs visible when maintaining a large test suite like this. Furthermore, the test cases are coded as PASS, XPASS, FAIL, and XFAIL, with XPASS indicating that the test case passed unexpectedly. When a test case passes unexpectedly, it means that the test case designer and developer are unaware of the feature's full functioning or the test case execution process. In such instances, the tester will be unable to verify feature completion or determine why a test case failed. In this example, assert statements would make a difference by informing the tester as to why the test case failed.

| A | B | C |
|---|---|---|
| Test File Name | Number of Assert Statements | Location of Assert Statements |
| ./gcc/testsuite/ada/acats/tests/c3/c354002.a | 83 | ['111', '126', '131', '132', '133', '135', '136', '138', '141', '142', '143', '147', '149', '150', '152', '154', '155', '171', '172', '174', '176', '177', '178', '181', '183', '184', '187', '188', '189', '190', '191', '192', '196', '198', '199', '202', '203', '204', '205', '206', '207', '217', '218', '219', '220', '221', '222', '235', '238', '241', '244', '247', '250', '251', '252', '253', '254', '255', '275', '278', '283', '286', '289', '290', '291', '292', '293', '294', '299', '304', '307', '308', '309', '311', '314', '315', '316', '326', '327', '328', '329', '330', '331'] |
| ./gcc/testsuite/ada/acats/tests/c4/c460008.a | 8 | ['64', '257', '261', '264', '271', '274', '277', '280'] |
| ./gcc/testsuite/ada/acats/tests/c7/c760009.a | 6 | ['447', '465', '479', '491', '505', '520'] |
| ./gcc/testsuite/g++.dg/lto/pr45679-1_1.C | 6 | ['32', '36', '39', '41', '55', '57', '73'] |
| ./gcc/testsuite/g++.dg/lto/pr45679-2_1.C | 6 | ['50', '54', '57', '59', '74', '76', '92'] |
| ./gcc/testsuite/gnat.dg/enum4.adb | 6 | ['5', '22', '26', '43', '47', '52'] |
| ./gcc/testsuite/ada/acats/tests/c4/c410001.a | 5 | ['242', '244', '246', '248', '284'] |
| ./gcc/testsuite/go.test/test/torture.go | 5 | ['198', '223', '249', '273', '297'] |
| ./gcc/testsuite/ada/acats/support/tctouch.ada | 4 | ['104', '105', '135', '142'] |
| ./gcc/testsuite/c-c++-common/builtin-arith-overflow- | 4 | ['34', '83', '94', '96', '103', '104', '353'] |
| ./gcc/testsuite/g++.dg/expr/sizeof2.C | 4 | ['25', '26', '28', '29'] |
| ./gcc/testsuite/g++.dg/ext/alias-decl-attr2.C | 4 | ['39', '40', '41', '42'] |

## Debug Statements in Prod Files

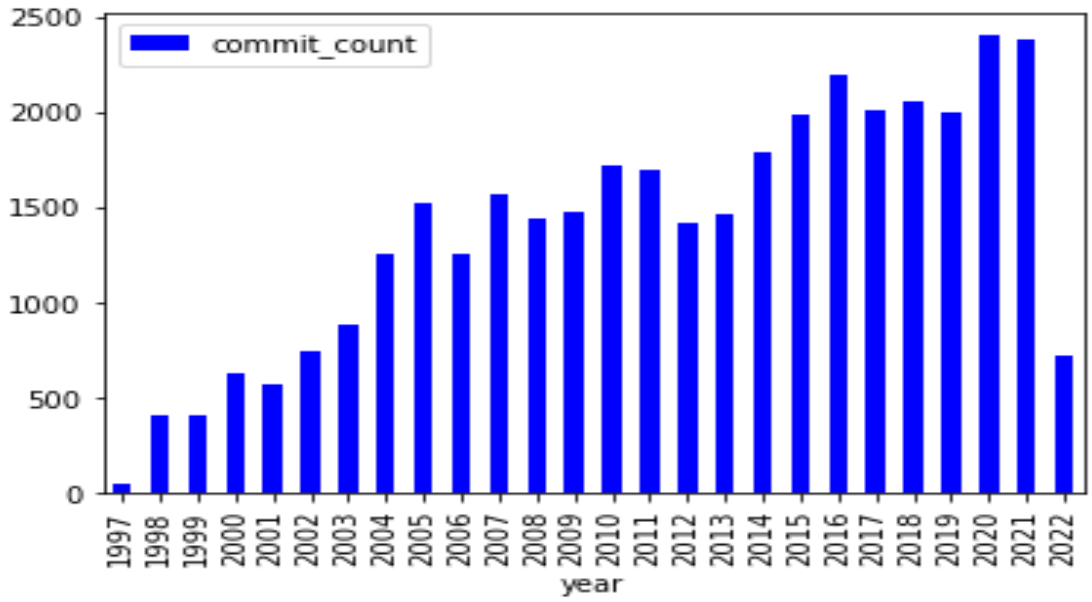
Maintaining debug statements in production files is a good and very good practice in general. When the application encounters problems, the debug statements assist us in understanding and analyzing the errors that arise. However, the ratio of debug statements to files is large enough to investigate a production problem. The image below shows a sample of the production files' debug statements.

B4 fx 23

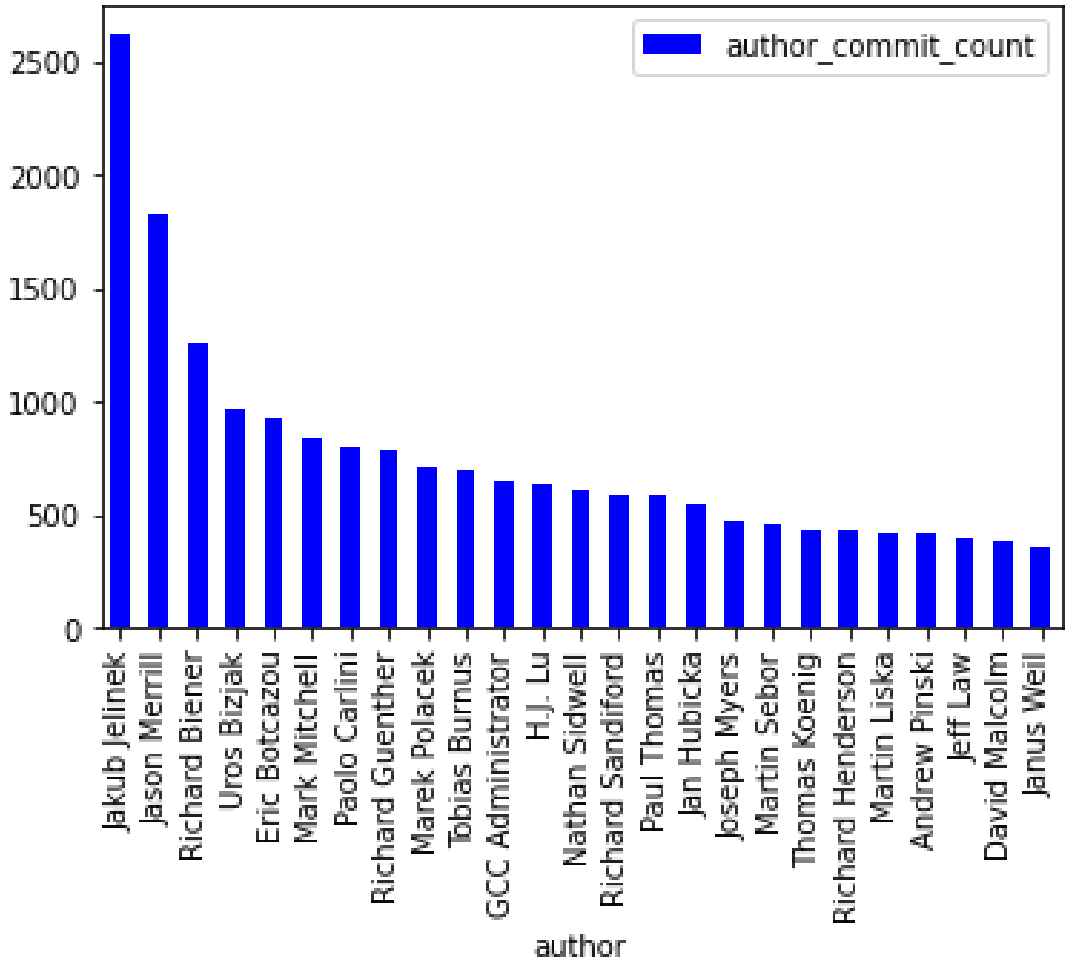
| File Name | Number of Assert Statements | Number of Debug State | Location of Assert Statements | Location of Debug Statements |
|---|---|---|---|---|
| ./libstdc++-v3/src/c++11/cow-stdexcept.cc | 3 | 0 | ['166', '168', '290'] | |
| ./gcc/fortran/trans-decl.cc | 77 | 2 | ['237', '247', '257', '326', '554', '572', '578', '699', '907', '990', '1107', '1108', '1133', '1136', '1229', '1312', '1343', '1395', '1409', '1602', '1623', '1896', '2125', '2249', '2255', '2271', '2378', '2562', '2638', '2640', '2679', '2815', '2823', '2844', '2850', '2921', '3024', '3174', '3279', '4129', '4130', '4156', '4284', '4290', '4574', '4800', '4809', '4980', '5041', '5116', '5138', '5181', '5185', '5187', '5207', '5208', '5261', '5328', '5341', '5537', '5570', '5613', '5718', '5735', '6018', '6019', '6020', '6021', '6127', '6128', '6129', '6214', '6610', '6963', '7546', '7552', '7830'] | ['5445', '5517'] |
| ./gcc/value-relation.cc | 23 | 1 | ['45', '46', '48', '61', '75', '89', '120', '121', '152', '153', '186', '187', '437', '530', '609', '824', '825', '826', '848', '871', '872', '922', '1366'] | ['1247'] |
| ./libgfortran/generated/matmul_c4.c | 10 | 0 | ['98', '260', '667', '829', '1236', '1398', '1819', '1981', '2461', '2623'] | |
| ./gcc/ada/exp_ch11.adb | 6 | 1 | ['120', '121', '888', '1406', '1989', '1990'] | ['961'] |
| ./gcc/d/dmd/denum.d | 2 | 0 | ['72', '290'] | |
| ./zlib/contrib/ada/zlib.adb | 2 | 0 | ['407', '408'] | |
| ./gcc/config/i386/i386.h | 1 | 2 | ['2277'] | ['2795', '2796'] |
| ./gcc/tree-loop-distribution.cc | 16 | 13 | ['417', '952', '989', '997', '1346', '1378', '1379', '1476', '2225', '2395', '2513', '2542', '2561', '2620', '2688', '3300'] | ['769', '912', '1014', '1021', '1023', '1049', |
| ./libgo/Makefile.am | 0 | 1 | | ['220'] |

## Commit analysis

The test suite was first implemented in 1997 and has since been updated. There has been a huge increase in adding/updating the test suite over the last few years. They've been adding test cases for many languages as well as compatibility with various host systems. The below image shows the year count for the folder gcc/testsuite.




Around 1000 developers have been contributing to the test suite over the years. The below image depicts the top 25 developers and their commit count for the repository gcc/testsuite



## Conclusion:

The fact that the project has roughly 190,000 commits and thousands of engineers contributing to it means that the test suite is constantly supported and updated. By adding test cases to boost coverage metrics, the test suite's sufficiency is maintained. However, assert statements for test cases are not very common, which makes analyzing test reports for failed test cases difficult.