# GCC Project

By

Sai Sree Mareddy

Arramareddy Mahitha Reddy

Burri Chaitanya Kumar Reddy

Submitted in fulfillment of the requirements for the course Program Analysis and Testing under the department of Master of Science in Computer Science

University of Houston

2022

# Table of Contents

# 1. GCC Project

## 1.1 Introduction

The correctness of compilers is crucial to the health and dependability of programming frameworks, because faults in compilers might result in executables that do not reflect the purpose of software engineers. Compiler testing typically employs arbitrary test program generators, which have proven to be successful in finding bugs. Nonetheless, the challenge of leading these test generators to create test programs that are certain to uncover problems remains.

Compiler testing is extremely tough due to their size and complexity. Testing such huge, modern frameworks like GCC is a major undertaking. Arbitrary testing, often known as fluffing, is a popular, lightweight method for building compiler test programs. Many programming languages offer arbitrary test generators that deliver test programs in those languages using the language syntax and language-explicit heuristics. Github Link: https://github.com/saisreemareddy/gcc-pat-assignment.git

## 1.2 GCC Source code

GCC stands for GNU Compiler collection, a set of compilers and development tools available in Linux, Windows, Various BSDs, and a wide range of other OS. It includes support primarily for C and C++ and includes Objective-C, Ada, Go, Fortran, and D. These magic three letters transform the C, C++, JAVA source code into a program that can be executable. The GCC compilation process has mainly four steps: preprocessor, compiler, assembler and linker.

Stage 1: In this stage, the preprocessor removes comments, includes header files (eg: <stdio.h>, <stdlib.h>), tokenizes the code and expands macros.

Stage 2: Next step, the compiler converts each line of preprocessed code into assembly language, which creates syntax trees of the program’s objects and control flow.

Stage 3: Now, Assembler transforms the compiled code to object code/machine language (pure binary code) that the computer can execute.

Stage 4: Finally, Linker extract functions contained in the program code from the library, then extract and connect them to output file. It also takes multiple files and merge into one program. It outputs the executable file with '.out' extension.

### 1.3 GCOV Coverage Analysis tool:

GCOV is a test coverage program. It is used in concert with GCC to analyse the programs to help create more efficient, faster running code and to discover untested parts of the program. You can use gcov as a profiling tool to help discover where optimization efforts will best affect the code. Gcov can also be used along with the other profiling tool, Gprof, to assess which parts of the code use the greatest amount of computing time.

Profiling tools helps in analyzing the code's performance. Using a profiler such as gcov or gprof, some basic performance statistics can be found out, such as:

· How often each line of code executes

· What lines of code are actually executed

· How much computing time each section of code uses

Once these things about how the code works when compiled are known, you can look at each module to see which modules should be optimized. Gcov helps to determine where to work on optimization. Gcov works only on code compiled with GCC. It is not compatible with any other profiling or test coverage mechanism.

### 1.4 GCC-Test Suite:

GCC has a large test suite: just for a small time, a large number of tests run covering several aspects of the compiler, such as:

- Handling invalid and valid syntax in the front-end.
- Verifying that the generated object code runs correctly.
- Checking for optimization passes
- Checking the working of debugger such as stepping through the code and can print metadata.

Especially, GCC using DejaGnu test suite from GCC 7. Which gives us ability to write our own test case files, can be annotated with magic comments: a domain-specific language for expressing how to compile the source files and the results. This makes relatively easy to turn an issue into a test case also popular as arbitrary testing or fluffing. But, Due to time-constraints, we couldn't utilize this feature and looking forward to trying this.

## 1.5 Installation and Build:

To build GCC source code, We had to install the Windows Wsl for Ubuntu 20.04.04 LTS. After git cloning the gcc source code repository, we checked out releases/gcc-11 to run configure and build this to avoid the unnecessary complicated errors. We installed the pre-requisites and build-essential to configure and flex module as per the guidelines in the GCC.wiki website.

./contrib/download_prerequisites

Sudo apt-get install build-essential

## 1.6 Reports:

Question :1 Line, Branch and Function coverage of the files in the source code

The source code has been built with enable coverage for coverage abilities and disable-multilib to run 64-bit libraries and --disable-werror. The build has taken a large part of the time because of the difficulties in understanding the errors and resolving them. We used Gcov tool to generate the coverage reports. The final coverage report for all the files has been included in the github repo. Here's a snapshot of it.

**GCC Code Coverage Report**

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

| | Exec | Total | Coverage |
|---|---|---|---|
| Lines: | 43550 | 983963 | 4.2% |
| 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 |
| gcc/gcc/analyzer/bar-chart.cc | | 0.0% | 0 / 32 | 0.0% | 0 / 3 | 0.0% | 0 / 16 |
| gcc/gcc/analyzer/bar-chart.h | | 0.0% | 0 / 3 | 0.0% | 0 / 2 | -% | 0 / 0 |
| gcc/gcc/analyzer/call-string.cc | | 0.0% | 0 / 78 | 0.0% | 0 / 10 | 0.0% | 0 / 40 |
| gcc/gcc/analyzer/call-string.h | | 0.0% | 0 / 6 | 0.0% | 0 / 5 | -% | 0 / 0 |
| gcc/gcc/analyzer/checker-path.cc | | 0.0% | 0 / 347 | 0.0% | 0 / 41 | 0.0% | 0 / 176 |
| gcc/gcc/analyzer/checker-path.h | | 0.0% | 0 / 85 | 0.0% | 0 / 38 | 0.0% | 0 / 12 |
| gcc/gcc/analyzer/complexity.cc | | 100.0% | 11 / 11 | 100.0% | 3 / 3 | 100.0% | 2 / 2 |
| gcc/gcc/analyzer/complexity.h | | 0.0% | 0 / 3 | 0.0% | 0 / 1 | -% | 0 / 0 |
| gcc/gcc/analyzer/constraint-manager.cc | | 0.0% | 0 / 800 | 0.0% | 0 / 62 | 0.0% | 0 / 632 |
| gcc/gcc/analyzer/constraint-manager.h | | 0.0% | 0 / 37 | 0.0% | 0 / 22 | 0.0% | 0 / 6 |
| gcc/gcc/analyzer/diagnostic-manager.cc | | 0.0% | 0 / 832 | 0.0% | 0 / 79 | 0.0% | 0 / 461 |
| gcc/gcc/analyzer/diagnostic-manager.h | | 0.0% | 0 / 9 | 0.0% | 0 / 6 | -% | 0 / 0 |
| gcc/gcc/analyzer/engine.cc | | 0.0% | 0 / 2106 | 0.0% | 0 / 165 | 0.0% | 0 / 1061 |
| gcc/gcc/analyzer/exploded-graph.h | | 0.0% | 0 / 174 | 0.0% | 0 / 77 | 0.0% | 0 / 66 |

We have coverage reports with line, function and branch coverage. As we can see, the coverage values are very low. But, this is only for a patch of tests. As the tests are taking enormous time and exhausting the memory and system resources.

## Question-2 and 3: Number of Assert and debug statements in all the files with their line number and path.

For generating the total number of assert statements in all the files and the line number of all assert statements in each test file is been done with the command of grep – linux command.

Ast.sh -----for generating the assert.txt with all the assert statements path and line number in each file. Code snippet

```
$ ast.sh
 1    grep -EcoR "^[^#--/*//\"].*Assert.*\(.*\)" ./gcc/testsuite/* | grep -v :0  >> assert.txt
```

For generating the total number of assert statements:

```
$ ast.sh
 1    grep -EcoR "^[^#--/*//\"].*Assert.*\(.*\)" ./gcc/testsuite/* | grep -v :0 | wc -l >> numasserts.txt
```

For generating all the debug statements in all files with path and line num:

```
$ dbg.sh
 1    grep -EcoR "^[^#--/*//\"].*Debug.*\(.*\)" ./gcc/testsuite/* | grep -v :0  >> dbg.txt
```

Similarly, for total number of debug statements:

```
$ dbg.sh
 1    grep -EcoR "^[^#--/*//\"].*Debug.*\(.*\)" ./gcc/testsuite/* | grep -v :0 | wc -l >> numdebug.txt
```

The output files are as follows:

Debug output snippet:

```
./gcc/testsuite/ada/acats/support/widechr.a:7
./gcc/testsuite/g++.dg/debug/pr46241.C:1
./gcc/testsuite/g++.dg/opt/combine.C:2
./gcc/testsuite/g++.dg/pr65240.h:2
./gcc/testsuite/g++.dg/torture/pr57235.C:2
./gcc/testsuite/g++.old-deja/g++.ext/pretty.C:1
./gcc/testsuite/gcc.c-torture/compile/pr52306.c:2
./gcc/testsuite/gnat.dg/debug10.adb:2
./gcc/testsuite/gnat.dg/debug4.adb:1
./gcc/testsuite/gnat.dg/loop_optimization4_pkg.adb:1
./gcc/testsuite/gnat.dg/loop_optimization4_pkg.ads:1
./gcc/testsuite/gnat.dg/nested_generic2_g1.adb:3
./gcc/testsuite/gnat.dg/nested_generic2_g1.ads:2
./gcc/testsuite/go.test/test/bench/garbage/parser.go:1
./gcc/testsuite/objc-obj-c++-shared/GNUStep/GNUstepBase/GSObjCRuntime.h:2
~
~
~
~
~
```

Assert statements output snippet:

```
./gcc/testsuite/ChangeLog-2019:1
./gcc/testsuite/ada/acats/support/tctouch.ada:4
./gcc/testsuite/ada/acats/tests/c3/c332001.a:3
./gcc/testsuite/ada/acats/tests/c3/c354002.a:83
./gcc/testsuite/ada/acats/tests/c3/c354003.a:23
./gcc/testsuite/ada/acats/tests/c3/c391002.a:20
./gcc/testsuite/ada/acats/tests/c3/c392004.a:11
./gcc/testsuite/ada/acats/tests/c3/c3a0005.a:2
./gcc/testsuite/ada/acats/tests/c3/c3a0007.a:2
./gcc/testsuite/ada/acats/tests/c4/c410001.a:5
./gcc/testsuite/ada/acats/tests/c4/c460008.a:8
./gcc/testsuite/ada/acats/tests/c4/c460011.a:11
./gcc/testsuite/ada/acats/tests/c4/c490001.a:2
./gcc/testsuite/ada/acats/tests/c4/c490002.a:2
./gcc/testsuite/ada/acats/tests/c6/c641001.a:14
./gcc/testsuite/ada/acats/tests/c7/c760001.a:16
./gcc/testsuite/ada/acats/tests/c7/c760002.a:31
./gcc/testsuite/ada/acats/tests/c7/c760009.a:6
./gcc/testsuite/ada/acats/tests/c7/c760011.a:2
./gcc/testsuite/ada/acats/tests/cb/cb40005.a:3
./gcc/testsuite/ada/acats/tests/cd/cd10001.a:1
./gcc/testsuite/ada/acats/tests/cd/cdb0a01.a:2
./gcc/testsuite/ada/acats/tests/cd/cdb0a02.a:2
./gcc/testsuite/ada/acats/tests/cxa/cxac005.a:13
./gcc/testsuite/ada/acats/tests/cxh/cxh3002.a:11
./gcc/testsuite/c-c++-common/attr-copy.c:1
./gcc/testsuite/c-c++-common/builtin-arith-overflow-2.c:4
./gcc/testsuite/c-c++-common/builtin-has-attribute-2.c:1
./gcc/testsuite/c-c++-common/builtin-has-attribute-3.c:1
./gcc/testsuite/c-c++-common/builtin-has-attribute-4.c:1
./gcc/testsuite/c-c++-common/builtin-has-attribute-5.c:1
./gcc/testsuite/c-c++-common/builtin-has-attribute-6.c:1
./gcc/testsuite/c-c++-common/builtin-has-attribute-7.c:3
./gcc/testsuite/c-c++-common/builtin-has-attribute.c:1
./gcc/testsuite/g++.dg/cpp0x/constexpr-arith-overflow.C:3
./gcc/testsuite/g++.dg/cpp0x/gen-attrs-17.2.C:1
./gcc/testsuite/g++.dg/cpp0x/gen-attrs-17.C:1
./gcc/testsuite/g++.dg/cpp1y/constexpr-arith-overflow.C:1
./gcc/testsuite/g++.dg/expr/sizeof2.C:4
./gcc/testsuite/g++.dg/ext/alias-decl-attr1.C:1
./gcc/testsuite/g++.dg/ext/alias-decl-attr2.C:4
./gcc/testsuite/g++.dg/ext/align2.C:1
./gcc/testsuite/g++.dg/ext/attrib17.C:1
./gcc/testsuite/g++.dg/ext/flexary.h:1
./gcc/testsuite/g++.dg/ext/tmplattr2.C:1
./gcc/testsuite/g++.dg/ext/tmplattr3.C:4
./gcc/testsuite/g++.dg/lto/pr45679-1_1.C:6
./gcc/testsuite/g++.dg/lto/pr45679-2_1.C:6
./gcc/testsuite/g++.dg/other/abstract1.C:3
```

## 2. PyDriller Tool:

### 2.1 Introduction:

PyDriller is a Python framework that helps developers on mining software repositories. With PyDriller you can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export CSV files. Moreover, PyDriller calculates structural metrics of every file changed in a commit relying on Lizard, a tool that can analyze source code of different programming languages, both at class and method level.

### 2.2 Installation:

The command used for installing pydriller :

#pip install pydriller

### 2.3 Reports:

In this section, We did the reports of all the commits in the gcc source code with the code in stats.py. We generated the commit number (hash), commit msg, commit author name and committed time. We also used the modified files to access all the changed files in a respective commit. The output has been generated in the stats.txt. For our own repository, we included the localStates.py, to get our commits with the same criteria and generated into the file local stats.txt. The code snippet is as follows:

```
from pydriller import Repository



for commit in Repository('https://github.com/saisreemareddy/gcc-pat-assignment.git').traverse_commits():
    print(commit.hash)

    print(commit.msg)
    print(commit.committer_date)

    print(commit.author.name)

    for file in commit.modified_files:
        print(file.filename, ' has changed')
```

## 3. Conclusion:

### 3.1 Conclusion:

In this Project, Development and testing larger projects is consider from their respective role point of view. Building project, performing line, branch and function coverage for all the file in the GCC source code. In the first, we have given the introduction to all the tools we have used in the project and included the reports and code snippets as per the requirements of the submission.

We also performed coverage analysis and assert statement reports for our own testcase return for testing GCC source compiler, which we have include in the sample work folder in our git repository.

### 3.2 Future Scope:

As we mentioned in the above, we are highly interested in continuing this project by doing arbitrary testing which generates test files out of issues reported before directly. Also, We would want to try the Deja GNU feature of adding our own test files to the source files for test suite.

## **4.** References


- https://gcc.gnu.org/install/
- https://gcc.gnu.org/install/test.html
- https://sodocumentation.net/gcc/topic/7873/code-coverage--gcov#:~:text=gcc%20Code%20coverage%3A%20gcov,-Remarks%23&text=Code%20coverage%20is%20a%20measure,tested%20by%20the%20test%20suite.
- https://pydriller.readthedocs.io/en/latest/intro.html