University of Houston

COSC 6386

Program Analysis and Testing

LLVM Project Report

Project Link: https://github.com/tennika20/PAT_Project_LLVM.git

Team Members:
Botta, Tennika Chowdary (PSID 2045101)
Challa, Varshitha (PSID 2156555)
Dammalapati, Divyasai (PSID 2154349)

INTRODUCTION

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. The name “LLVM” itself is not an acronym; it is the full name of the project. LLVM began as a research project with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research.

Some of the primary sub-projects of LLVM are:
LLVM Core Libraries: These provide a modern source and target independent optimizer, along with code generation support for many popular CPUs . These libraries are build around a well specified code representation known as LLVM intermediate representation LLVM IR.

Clang: This is an LLVM native C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great soure level tools.

LLDB Project: The project builds on libraries provided by LLVM and Clang to provide a great native debugger. It uses the Clang ASTs and expression parser, LLVM JIT, LLVM disassembler, providing an experience that just works.

Compiler-rt: The compiler-rt project provides highly tuned implementations of the low-level code generator support routines and other calls generated when a target doesn’t have a short sequence of native instructions to implement a core IR operation. It also provides implementations of run-time libraries for dynamic testing tools.

MLIR: This subproject builds reusable and extensible compiler infrastructure aiming for address software fragmentation, improve compilation for heterogenous hardware, significantly reduce the cost of building domain, specific compilers, and aid in connecting existing compilers together.
And OpenMP, polly, libclc, klee, LLD projects.

THE SOFTWARE VIEW:

Any software development firm’s ultimate purpose is to produce high-quality software. The software must be carefully tested in order to accomplish this goal and as a software engineer, we would be working in collaboration with developers and testers and come up with all the possible test cases. Based on time frame and budget a proper end to end test management plan should be defined with all entry and exit criteria. Testing is done at various levels in most of the time, the

five tiers are Unit testing, Integration Testing, System testing, Acceptability Testing, and Regression Testing. Each level of testing has it's own set of testing goals. At the Unit testing level, for example, functional and/ or structural testing methodologies are used to test independent units. Two or more units are merged at the integration testing level, and testing is performed to test the integration issues of the individual units. The system is tested as a whole at the system testing level, with functional testing methodologies being employed predominantly. Non functional requirements like performance, reliability, usability, testability etc., are also tested at this level. Testing can be done at different levels with the aid of software testing tools, whenever we have a failure, we debug the source code to figure out what really happened. Finding the causes of a failure is time consuming testing task that requires a lot of resources and may cause the program to be delayed. Hence, it is important to practice good programming standards which makes it easy to debug in the case of any code breakages.
To overcome this problem, we can make use of assert and debug statements to make it easy for the developers to check and fix the errors at the earliest possible. An effective way to examine if the testing of any application is done properly or not is Code Coverage. This helps developers and testers to make sure that the test cases cover every line, branch and function in the source code and allows them to build code and test suites accordingly which leads to a good quality product. As mentioned above, in this report we have included coverage report, count of assert and debug statements in the test and production files for the LLVM project. We have also retrieved the details of the people who worked on the project in a particular time period using pydriller.

1. PROJECT SETUP:

Compiling LLVM requires the following packages installed. The Package column is the usual name for the software package that LLVM depends on. The Version colum provides "known to work" versions of the package.The Notes column describes how LLVM uses the package.

Package	**Version**	**Notes**
CMake	>=3.13.4	Makefile/workspace generator
GCC	>=7.1.0	C/C++ compiler
python	>=3.6	Automated test suite
GNU Make	3.79	Makefile/build processor

2. BUILDING LLVM:

Clone the project repo: git clone https://github.com/llvm/llvm-project.git
cd llvm-project
mkdir build
cd build
cmake -G <generator> -DCMAKE_BUILD_TYPE=<type> [options] ../llvm

Some common build system generators are:

- Ninja — for generating ninja build files. Most llvm developers use Ninja.
- Unix Makefiles — for generating make-compatible parallel makefiles.
- Visual Studio — for generating Visual Studio projects and solutions.
- Xcode — for generating Xcode projects.

Some Common options:

- -DLLVM_ENABLE_PROJECTS='...' — semicolon-separated list of the LLVM subprojects you'd like to additionally build.
  For example, to build LLVM, Clang, libcxx, and libcxxabi, use -DLLVM_ENABLE_PROJECTS="clang" -DLLVM_ENABLE_RUNTIMES="libcxx;libcxxabi".
- -DCMAKE_INSTALL_PREFIX=directory — Specify for *directory* the full pathname of where you want the LLVM tools and libraries to be installed (default /usr/local).
- -DCMAKE_BUILD_TYPE=type — Controls optimization level and debug information of the build. The default value is Debug which fits people who want to work on LLVM or its libraries. Release is a better fit for most users of LLVM and Clang.
- -DLLVM_ENABLE_ASSERTIONS=On — Compile with assertion checks enabled (default is Yes for Debug builds, No for all other build types).

cmake - - build . [- - target <target>] or the build system specified above directly.

3. TESTING LLVM:

LLVM infrastructure uses the following commands in order to run the tests.

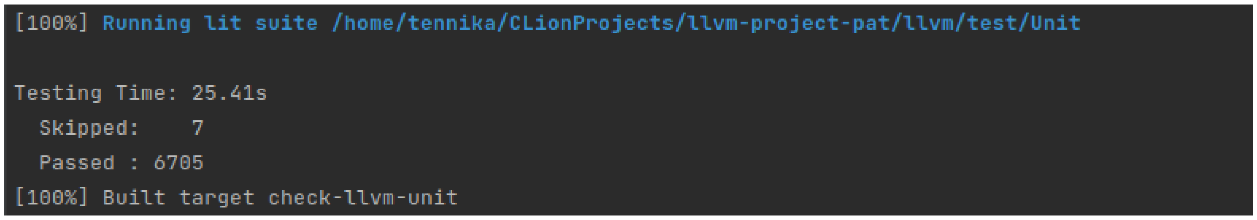
make check-llvm-unit*:* runs all of the LLVM unit tests

make check-llvm*:* runs all of the LLVM regression tests

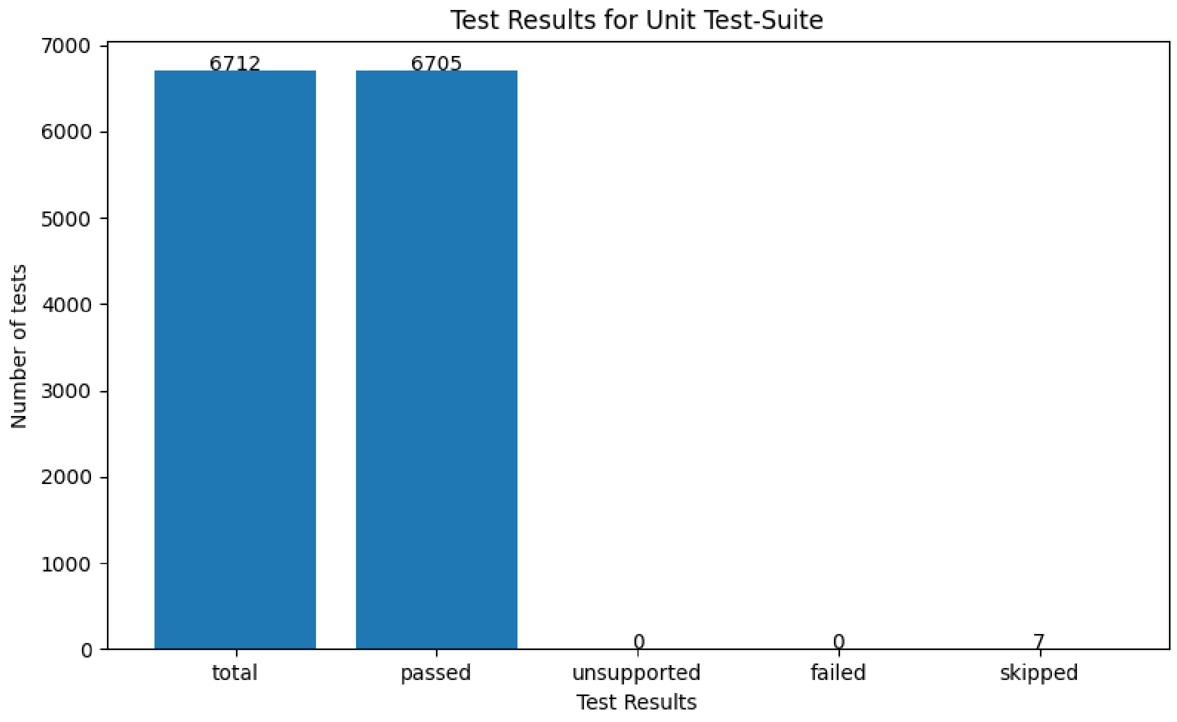
TEST ANALYSIS

Unit Tests:

The unit tests are located in the llvm/ unittests directory. These are reserved for targeting the support library and other generic data structure.

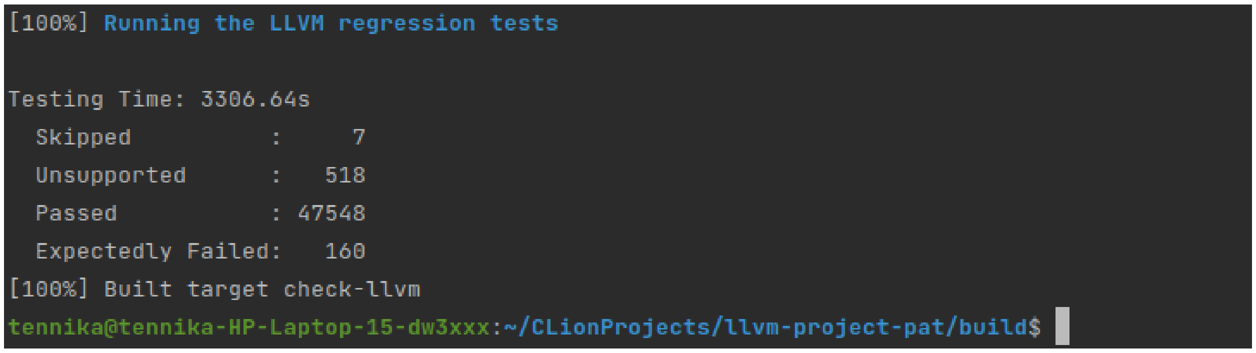


```
[100%] Running lit suite /home/tennika/CLionProjects/llvm-project-pat/llvm/test/Unit

Testing Time: 25.41s
  Skipped:    7
  Passed : 6705
[100%] Built target check-llvm-unit
```

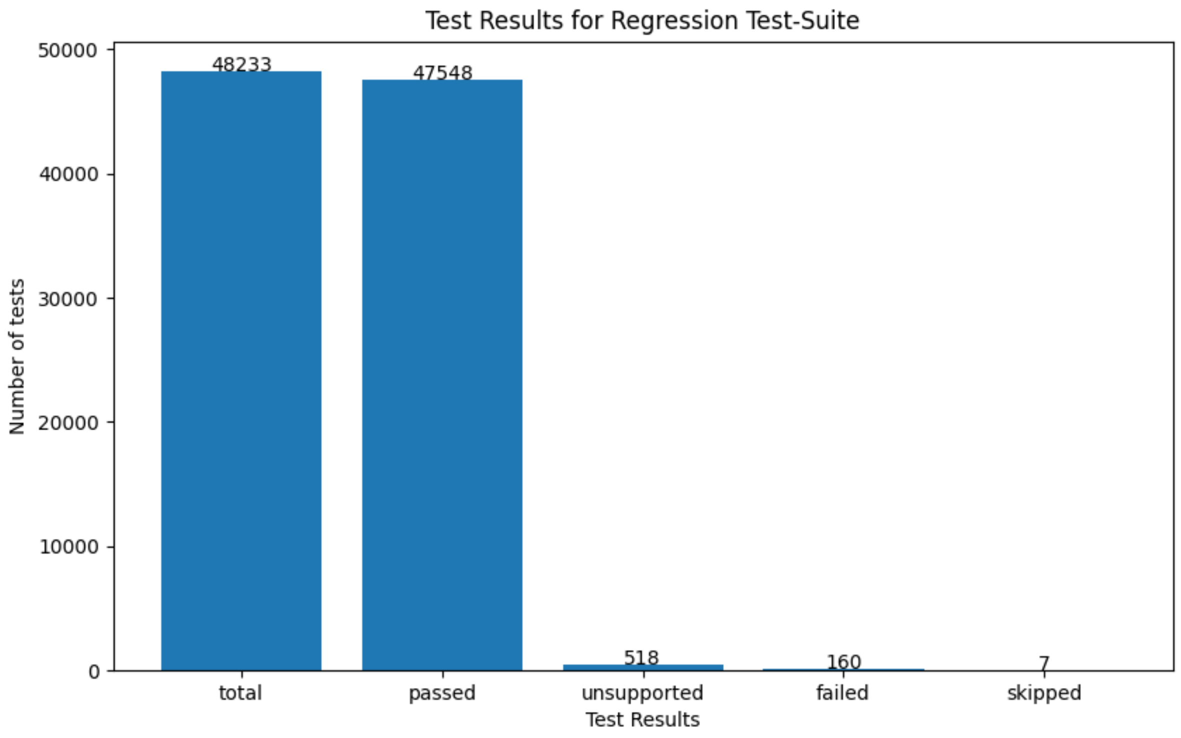


Regression Tests:

The regression tests are small pieces of code that tests a specific feature of LLVM or trigger a specific bug in LLVM. These tests are driven by the lit testing tool, and are located in the llvm/test directory.



```
[100%] Running the LLVM regression tests

Testing Time: 3306.64s
  Skipped          :     7
  Unsupported      :   518
  Passed           : 47548
  Expectedly Failed:   160
[100%] Built target check-llvm
tennika@tennika-HP-Laptop-15-dw3xxx:~/CLionProjects/llvm-project-pat/build$
```



4. ADEQUACY CRITERIA OF THE TESTS

A test adequacy criteria is a predicate that is true (satisfied) or false (not satisfied) of a program and a test suite pair. This test adequacy is expressed in the form of a rule for deriving a set of test obligations from another artifact, such as a program or a specification.

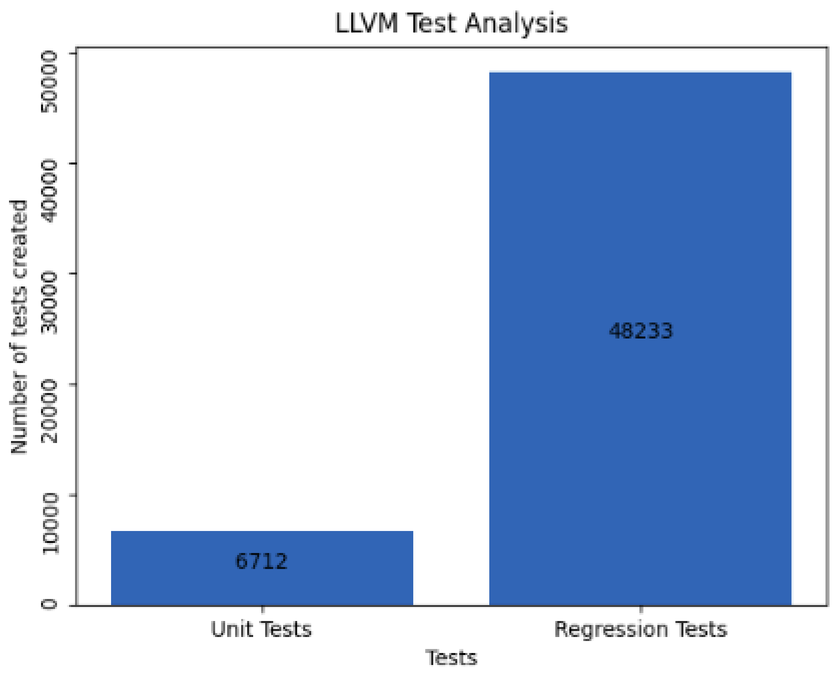
From the tester's point of view, the adequacy criteria is coverage and in developer's perspective, there is no test suite that can meet the test adequacy criteria for a program.

The developers test the structural and path coverage in this testing to influence the adequacy criterion. For the structural coverage, that covers things like loops, pathways and dataflow, the loop requires sufficient test cases for program loops to be executed for zero, one, two and many iterations covering initialization, typical running and termination. In the loop coverage, one of the things that needs to test is to make sure that at some point, the loop does not get locked and the program continues to execute on into infinity, that it should terminate at a point.

The path coverage, it requires the test cases for each feasible path, basis path, from start to exit of a defined program segment, again to be executed at least once to show that it will work. Because of the very large number of possible pathways through the program, path converge is generally achievable, from the standpoint that there may not be too many variables again.

5. APPROPRIATENESS OF TESTING PROCESS

The LLVM Project takes about 8 hours for a build and the testing takes about 45 minutes for both the unit and regression tests. LLVM is an ample size of project, and hence it requires good amount of tests to verify the steadiness of the project. As it consumes a lot time for the initial build and test process, these responsibilities can be automated. If there is any failure occurred during build, it gets terminated and the tester has to find the bug in order to execute the build. But, in developer view, this process needs less attention and automation since it is a menial task.



6. CODE COVERAGE:

The code coverage for LLVM is as shown below

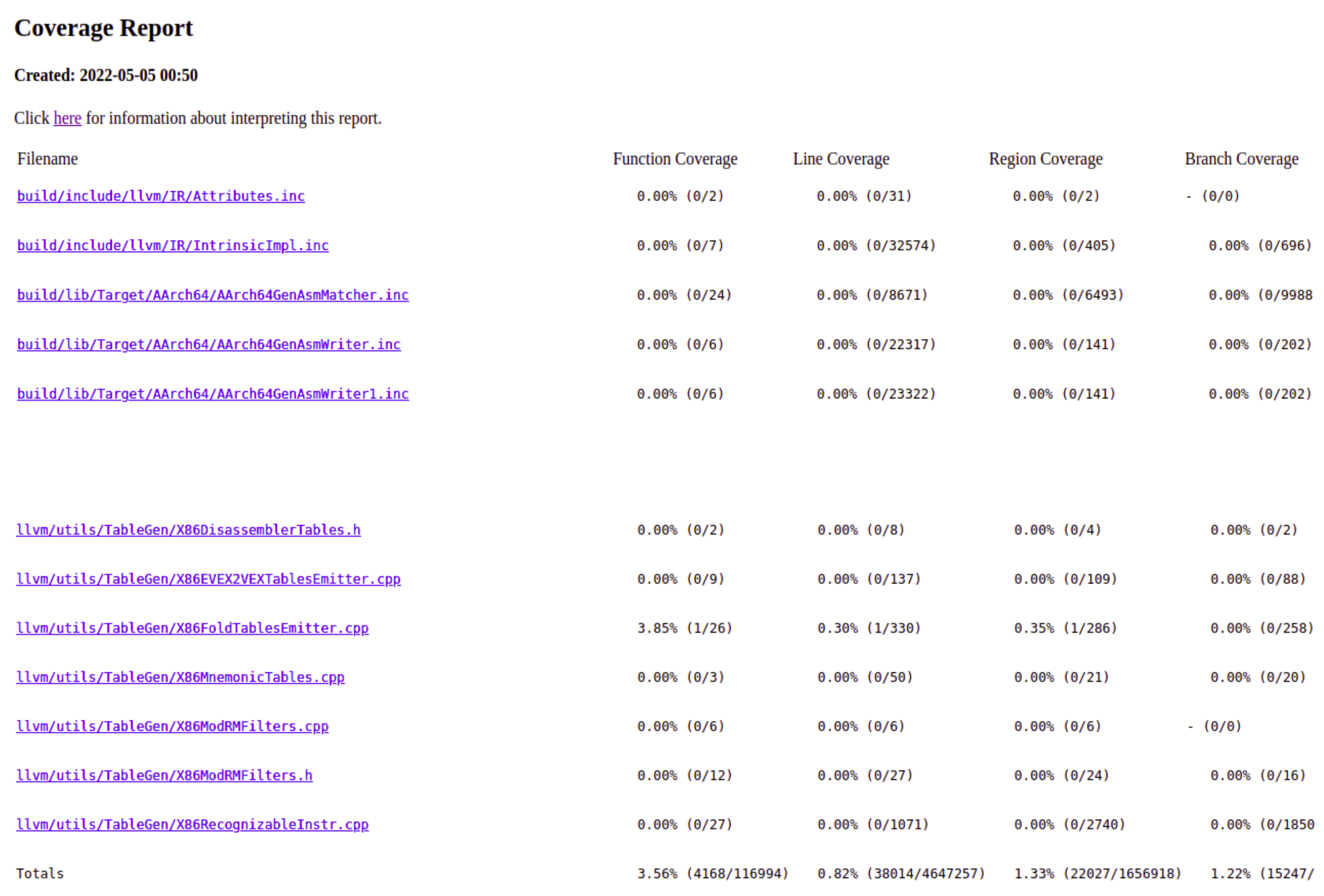
- **Function Coverage: 3.56%** (4168/116994)
- **Line Coverage: 0.82%** (38014/4647257)

- **Region Coverage: 1.33%** (22027/1656918)
- **Branch Coverage: 1.22%** (15247/1250162)

For the generation of coverage report, we enabled the LLVM variable in HandleLLVMOptions.cmake file.

LLVM_BUILD_INSTRUMENTED_COVERAGE

This generated a generate-coverage-report directory in build/CMakeFiles folder. The build.make file contains command to run prepare-code-coverage-artifact.py



Coverage Report

Created: 2022-05-05 00:50

Click here for information about interpreting this report.

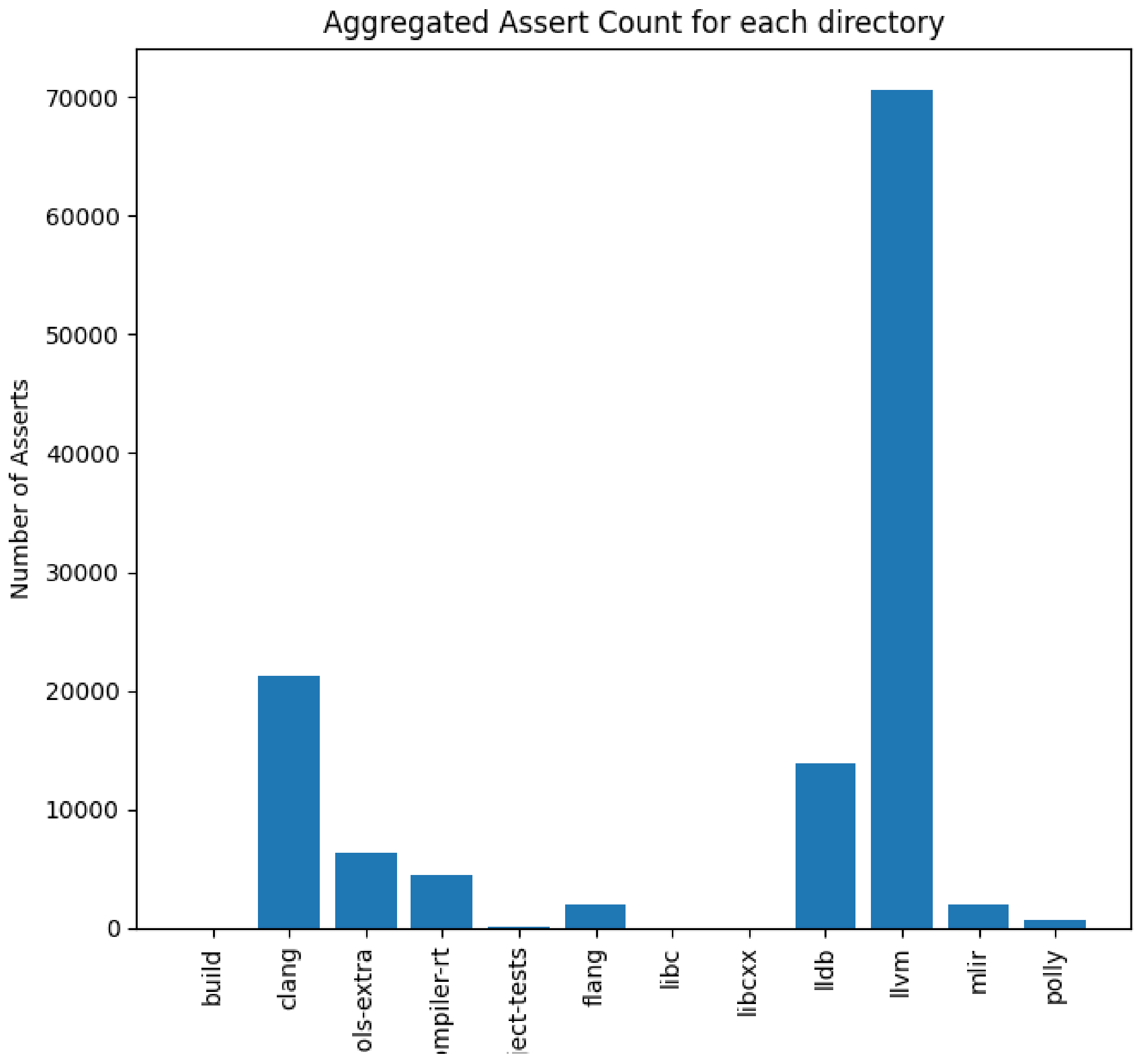
Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
build/include/llvm/IR/Attributes.inc	0.00% (0/2)	0.00% (0/31)	0.00% (0/2)	- (0/0)
build/include/llvm/IR/IntrinsicImpl.inc	0.00% (0/7)	0.00% (0/32574)	0.00% (0/405)	0.00% (0/696)
build/lib/Target/AArch64/AArch64GenAsmMatcher.inc	0.00% (0/24)	0.00% (0/8671)	0.00% (0/6493)	0.00% (0/9988
build/lib/Target/AArch64/AArch64GenAsmWriter.inc	0.00% (0/6)	0.00% (0/22317)	0.00% (0/141)	0.00% (0/202)
build/lib/Target/AArch64/AArch64GenAsmWriter1.inc	0.00% (0/6)	0.00% (0/23322)	0.00% (0/141)	0.00% (0/202)
llvm/utils/TableGen/X86DisassemblerTables.h	0.00% (0/2)	0.00% (0/8)	0.00% (0/4)	0.00% (0/2)
llvm/utils/TableGen/X86EVEX2VEXTablesEmitter.cpp	0.00% (0/9)	0.00% (0/137)	0.00% (0/109)	0.00% (0/88)
llvm/utils/TableGen/X86FoldTablesEmitter.cpp	3.85% (1/26)	0.30% (1/330)	0.35% (1/286)	0.00% (0/258)
llvm/utils/TableGen/X86MnemonicTables.cpp	0.00% (0/3)	0.00% (0/50)	0.00% (0/21)	0.00% (0/20)
llvm/utils/TableGen/X86ModRMFilters.cpp	0.00% (0/6)	0.00% (0/6)	0.00% (0/6)	- (0/0)
llvm/utils/TableGen/X86ModRMFilters.h	0.00% (0/12)	0.00% (0/27)	0.00% (0/24)	0.00% (0/16)
llvm/utils/TableGen/X86RecognizableInstr.cpp	0.00% (0/27)	0.00% (0/1071)	0.00% (0/2740)	0.00% (0/1850
Totals	3.56% (4168/116994)	0.82% (38014/4647257)	1.33% (22027/1656918)	1.22% (15247/

From the above report, we observe that the function coverage is 3.56%, line coverage is 0.82%, region coverage is 1.33% and branch coverage is 1.22%. This is a very low coverage, because of less number of tests. However, there can be several unrelated files that can result in the low coverage of the project.

7. ASSERT STATEMENTS IN EACH TEST FILE

Assert statements are assertions that are used to put assumptions to the test. It is a specification that confirms that a program meets certain requirements at specific points during execution (run-time condition checks). If the assert condition is false, an error will be generated, and our program will terminate with appropriate error messages. This not only assures that we want the program to perform what we want, but it also organizes the code in a logical and easy-to-read manner. Assert statements helps us to make sure that any design flaws or defects in our program are detected and handled appropriately.

In our project, we have multiple assert statements in test files depending upon the functionality that is being tested. The unit test's purpose is to provide us with as much as information as possible about what is failing while also assisting us in pointing the most fundamental issues first.



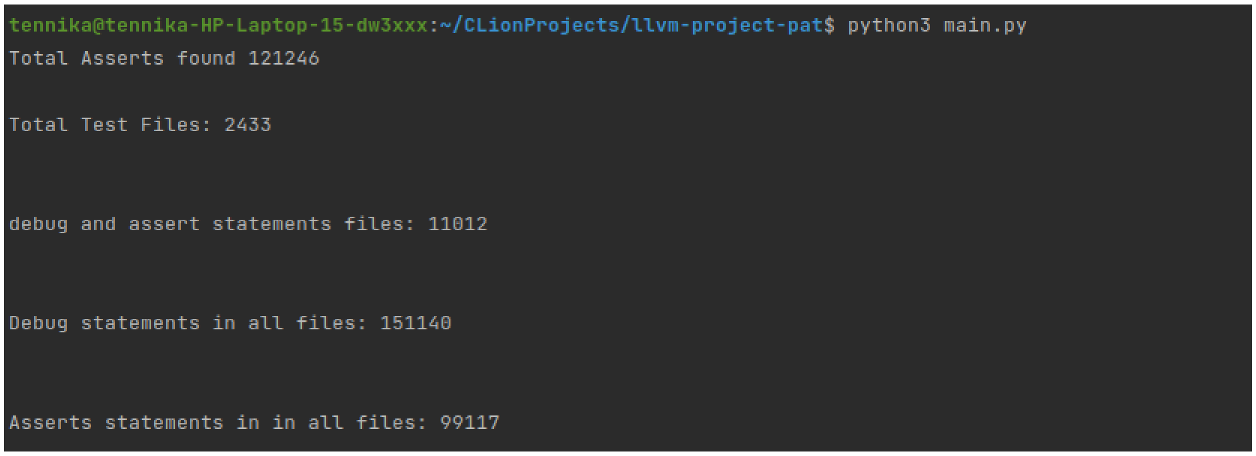
Above is the graph depicting the count of assert statements in each test file directory.

We have the names of the directories on x-axis and the corresponding count of assert statements on the y-axis. (since there are large number of test files containing assert statements, we have displayed the directories)

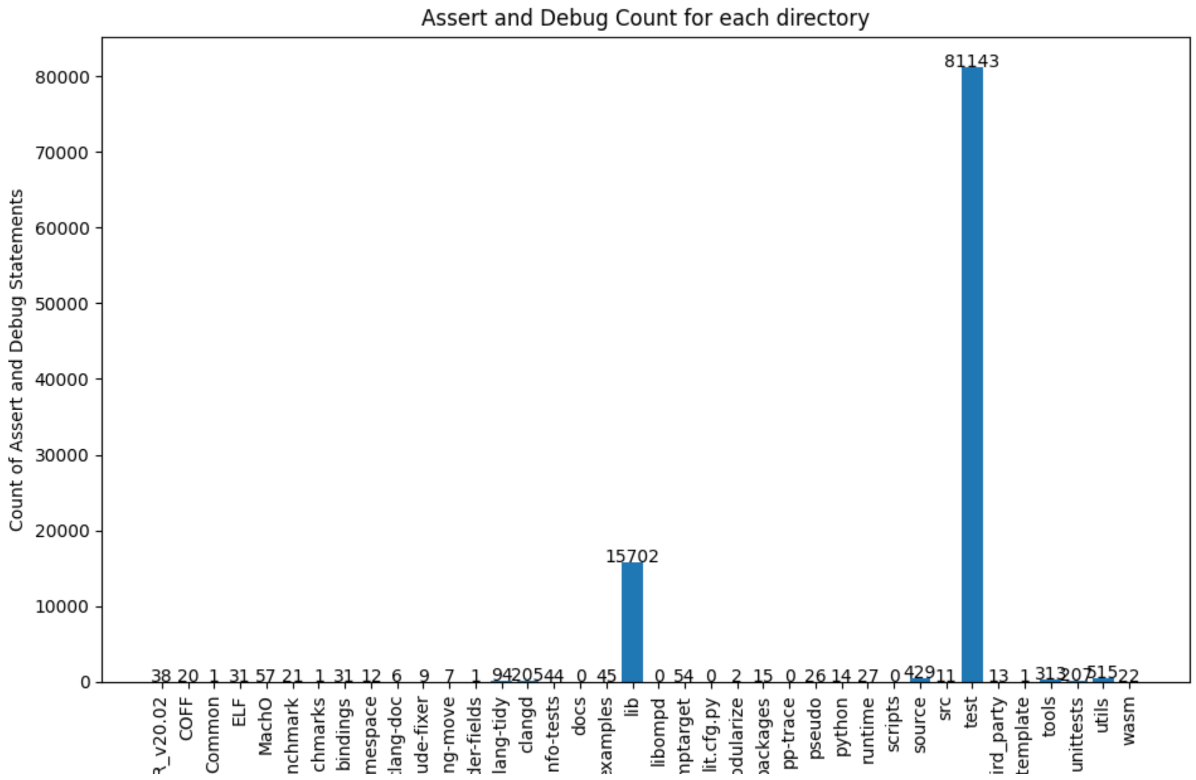
8. DEBUG AND ASSERT STATEMENTS IN PRODUCTION FILES

The debug and assert statements are found in production files. Since the developer predefines halting or diverting the execution of program based on particular criteria, assertion statements can be considered as active debugging. They will help us ensure that we don't institute any new bugs while adding features and fixing other bugs in our code. When we want to know why a piece of software fails we would try to identify a solution and make necessary adjustments to the source code in order to resolve the cause of software failure. The process of locating and repairing a software error is known as Debugging.

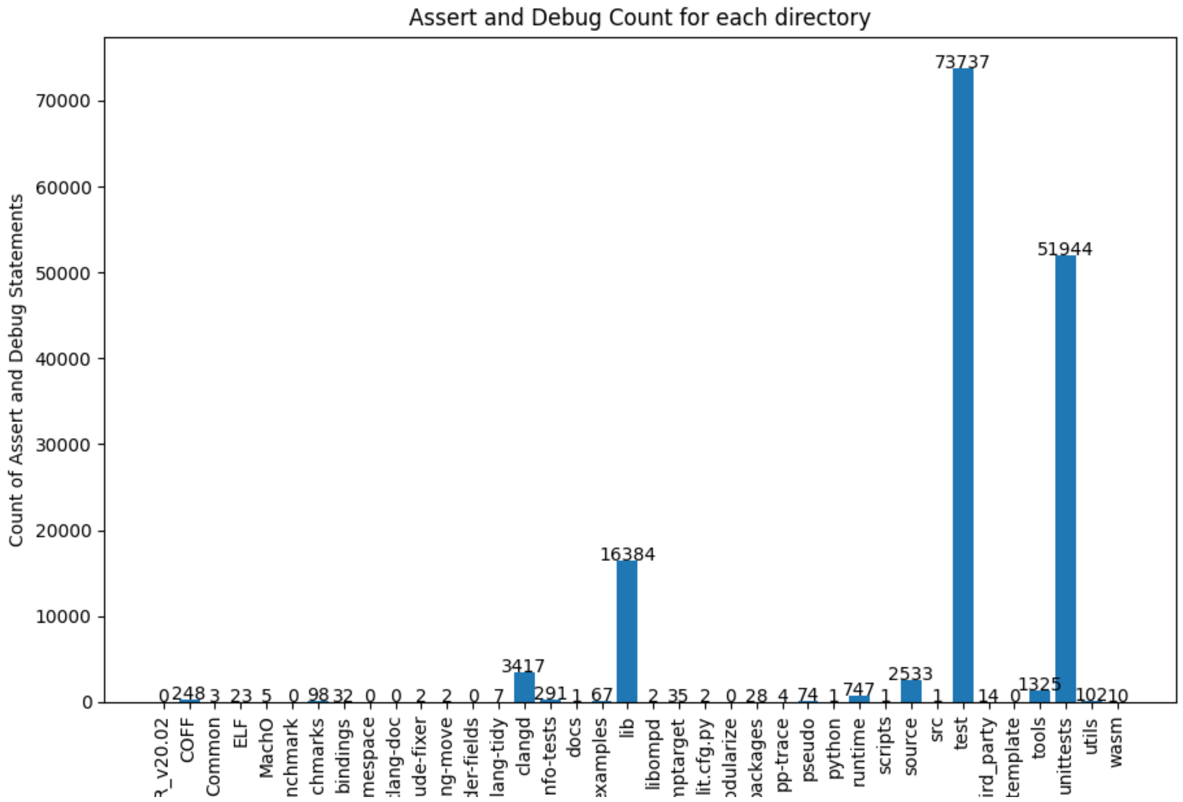
For this part of the project, we have located and calculated the number of debug and assert statements present in the production files of our project.



```
tennika@tennika-HP-Laptop-15-dw3xxx:~/CLionProjects/llvm-project-pat$ python3 main.py
Total Asserts found 121246

Total Test Files: 2433


debug and assert statements files: 11012


Debug statements in all files: 151140


Asserts statements in in all files: 99117
```



Assert and Debug Count for each directory
Count of Assert and Debug Statements
81143
15702



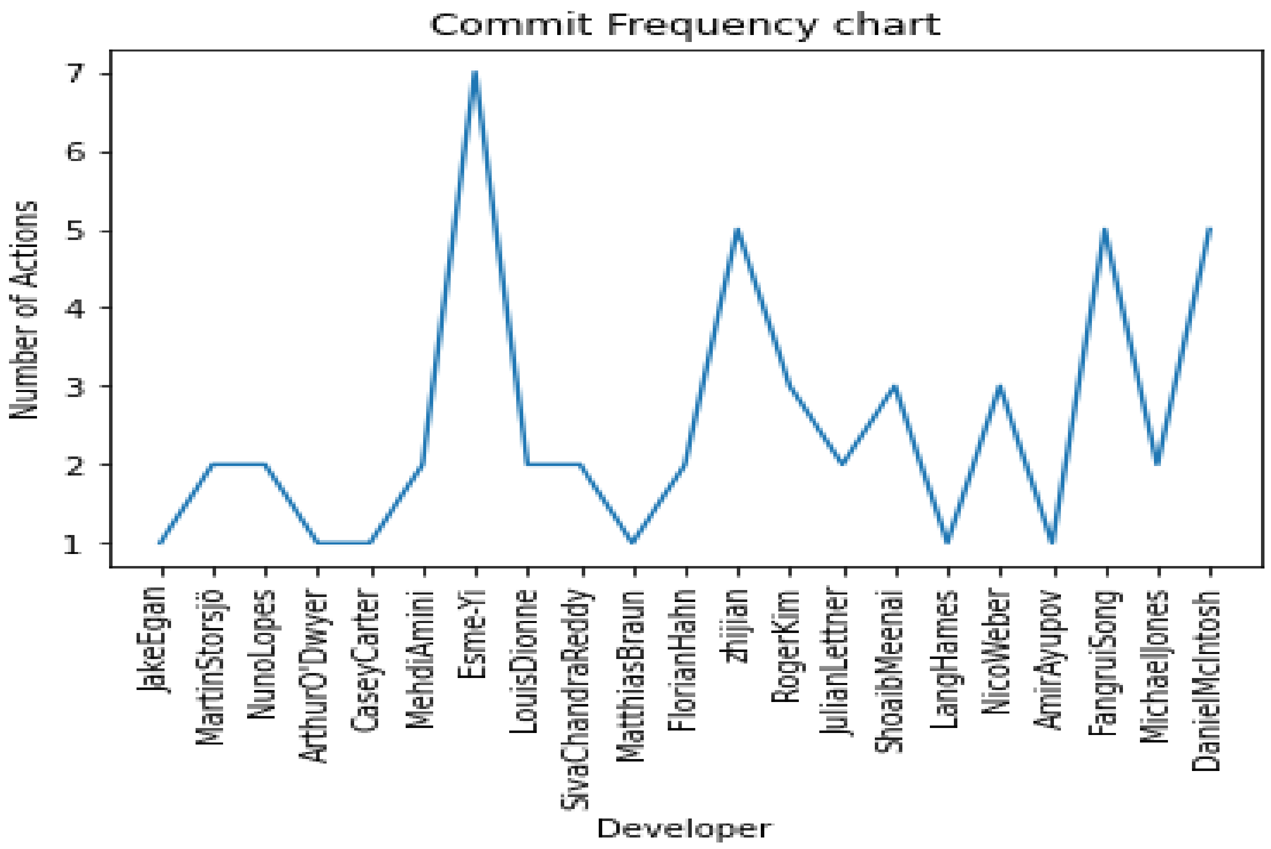
Assert and Debug Count for each directory
Count of Assert and Debug Statements
73737
51944
16384
3417
2533
1325
747

The above plot shows the count of assert and debug statements in production files. (The files that are not present in the test folders are considered as production files)

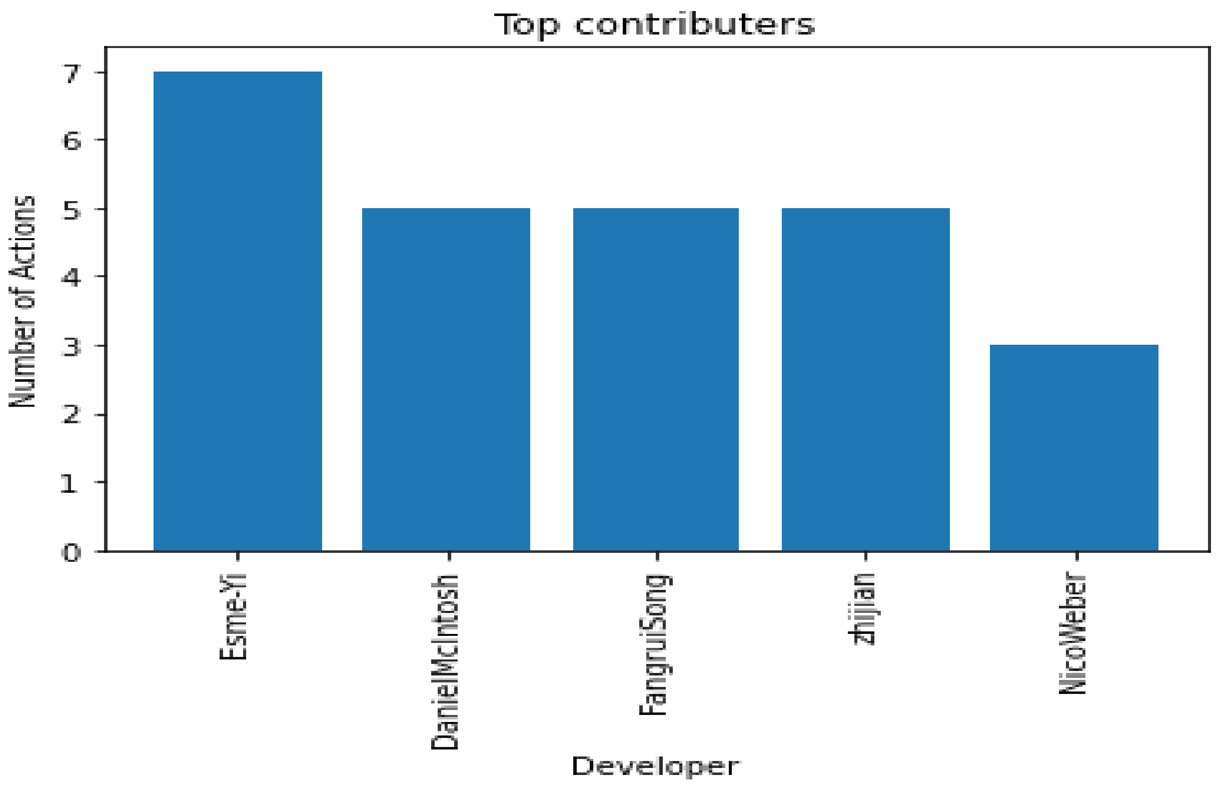
9. GENERATING COMMIT DETAILS USING PYDRILLER

PyDriller is a python framework that helps developers on mining the software repositories. With PyDriller one can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export CSV files.

The graph on Commit frequency for the past 5 months is generated as shown below



Since the data is large, we have considered for certain time period, i.e., past 5 months. Here, we can see the authors who made commits to the repository in recent times. We have also extracted top 5 contributors in the past 5 months.



CONCLUSION:

The LLVM project is built successfully, and analysed the testing process with few inferences. The code coverage report for the project is very low, which signifies that there isn't enough number of tests provided by the developers. Being a project of ample size, requires more number of contributions for the testing in order to generate higher code coverage. We have extracted and visualized assert statements in the test files and assert and debug statements in the production files for better analysis. Also we have derived the number of commits from the LLVM repository using PyDriller and visualized for a time period of 5 months. This information represents the modifications and other actions made by the authors in the past 5 months. We have also obtained the plot for top 5 contributors using PyDriller.

REFERENCES:

https://llvm.org/docs/GettingStarted.html

https://github.com/llvm-mirror/llvm

https://github.com/llvm/llvm-project

https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

https://llvm.org/docs/CommandGuide/llvm-cov.html