Program Analysis and Testing

COSC 6386 Spring 2022

Project

UNIVERSITY of HOUSTON

LLVM

GitHub Link: https://github.com/suryaKurella/LLVM_PAT_Project.git

Team:

Suryateja, Kurella (2050296)

Revathi Bhavani, Batchu (2148453)

Vikram, Pillarisetty (2093976)

Introduction

The **LLVM** (**Low Level Virtual Machine**) **Project** is a collection of reusable and modular compilers and toolchains. LLVM originated as a University of Illinois research project with the purpose of developing a modern, SSA-based compilation technique capable of enabling both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has evolved into an umbrella project with several subprojects, many of which are utilized in production by a wide range of commercial and open-source projects, as well as in academic research.

Building LLVM

LLVM is a huge compiler framework with several dependencies. Building the project is a key step for executing the test cases, installing several LLVM subprojects and generating code coverage.

For building this project, the following are the basic requirements:

Operating System	Windows/ Linux/ Mac/ Any Unix supported OS
CMake	>= 3.13.4
GCC	>= 7.1.0
Python	>= 3.6
GNU Make	3.79, 3.79.1

The following steps build LLVM:

- **git clone** https://github.com/llvm/llvm-project.git
- **cd llvm-project**
- **cmake -S llvm -B build -G <generator>[options]**
  - In this command -G indicates the type of generator, following are the generators supported by LLVM:
    - Ninja
    - Unix Makefiles
    - Visual Studio
    - Xcode
  - Generally, developers prefer Ninja build as it is generated quickly that other build types, we used **Unix Makefiles** to generate the build.
- **cmake –build build [--[options] <target>]**
  - This command builds the whole project from the build configuration provided in the above step.
- Following are some common options to provide while building LLVM:

  - **-DLLVM_ENABLE_PROJECTS** – builds the specified LLVM subprojects
  - **-DCMAKE_INSTALL_PREFIX** – installs LLVM in this specified directory
  - **-DCMAKE_BUILD_TYPE** – type such as Debug, Release, RelWithDebInfo and MiniSizeRel.
  - **-DLLVM_ENABLE_ASSERTIONS** – Compile with assertion checks enabled.
- We built the entire project in Debug Mode, which is the default build type that enables assertions, debug logs but with a reasonable performance overhead.

Testing in LLVM

LLVM Testing Infrastructure is built in such way that it is simple to run the unit and regression tests using the following commands.

- **make check-llvm-unit**
  - This command runs all the LLVM unit tests.
- **make check-llvm**
  - This command runs all the LLVM regression tests.

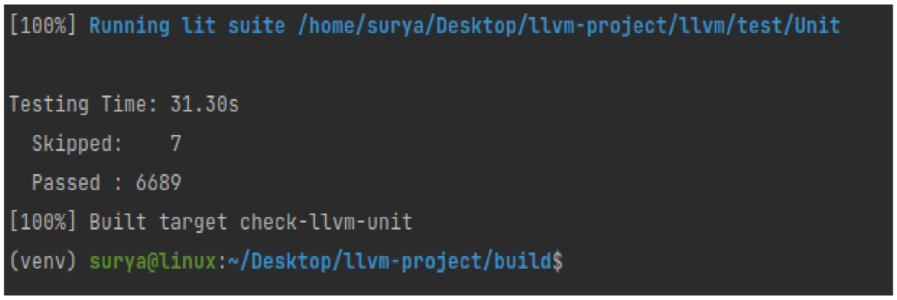
Alternatively, **LIT** tool can be used to run directory specific or individual test cases.

We used **"make check"** commands to run regression and unit tests at once instead of LIT.

Testing State Analysis of LLVM

The state of testing has followed different phases all over the project implementation. In the initial years from **2001** to **2010**, state of testing is moderate, later it was much effective from the year **2011** till **2019**. Thorough analysis of the state of testing is elucidated in the following sections.

Upon our analysis, the following are the test results obtained when the **Unit test cases** are run, there are about **6689** passed unit test cases and only **7** are skipped.

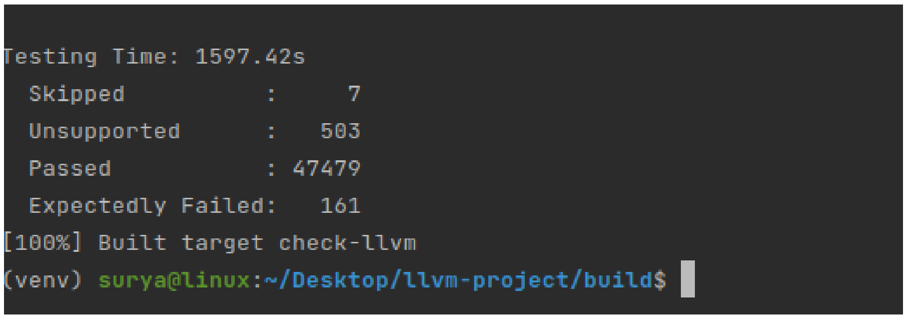


```
[100%] Running lit suite /home/surya/Desktop/llvm-project/llvm/test/Unit


Testing Time: 31.30s
  Skipped:    7
  Passed : 6689
[100%] Built target check-llvm-unit
(venv) surya@linux:~/Desktop/llvm-project/build$
```

Fig 1. Unit Test Stats

The following are the test results obtained when the **Regression test cases** are run. There are about **48,150** regression test cases, out of which **47479** test cases passed and **7** skipped.



```
Testing Time: 1597.42s
  Skipped          :     7
  Unsupported      :   503
  Passed           : 47479
  Expectedly Failed:   161
[100%] Built target check-llvm
(venv) surya@linux:~/Desktop/llvm-project/build$
```

Fig 2. Regression Test Stats

There are **161** test cases that are Expectedly failed, this could be because of false negatives, false positives, or errors due to environment or setup issues and failures due to fragile or flaky test automation. There are **503** unsupported cases which can only be executed under certain configurations, such as with debug builds or on specified platforms. To control when the test is enabled, we can use UNSUPPORTED target while building the project.

The detailed analysis of LLVM testing can be explained by Adequacy and Appropriateness of Tests which are explained in both developer's and tester's perspective.

Adequacy of Tests

Adequacy is a metric used to judge whether a test set meant to test S meets its requirements. This measurement is made in relation to a predetermined criterion C. When a test set meets C, it is called adequate in terms of criterion C. Whether or not a test set T for program S satisfies criterion C is determined by the criterion itself.

Adequacy of test is measured in 2 constraints:

- A test case T is adequate if it satisfies all the functional requirements R of the program.
  - This kind of testing is performed by a tester and often called black-box testing.
- A test case T is adequate if each path in P is traversed at least once.
  - This kind of testing is performed by a developer and often called white-box testing.

After the development, unit tests are performed by the developer where a unit or block of code is tested and let the developers know their code bugs. Mainly, the developer tests the path coverage in this testing and determines the adequacy of tests.

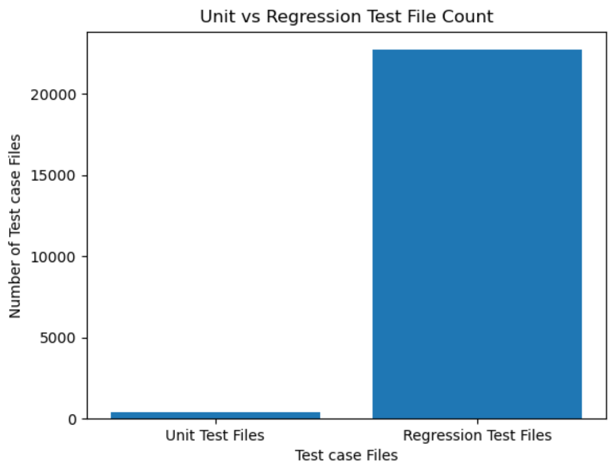


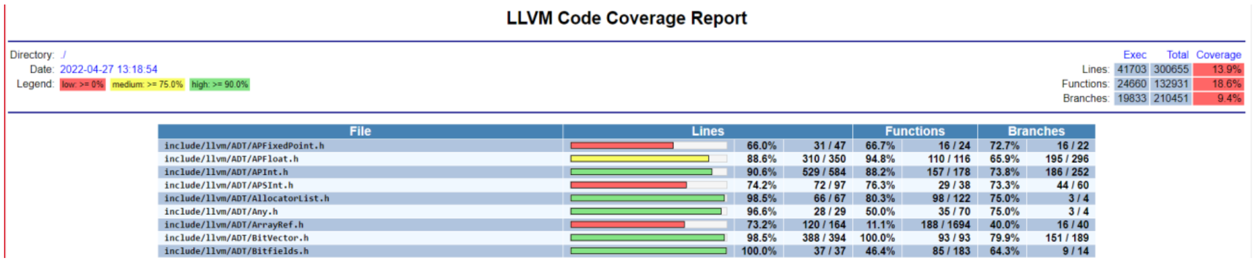
Fig 3. Unit vs Regression Test Files Count

In this project, there are about **6689-unit tests** created by the developers when compared to the **48,150 regression test cases** created by the testers. The following information explains the code coverage and concludes the adequacy of tests.

Code Coverage

Code coverage for LLVM can be explained by the following 3 factors:

- **Line coverage – 13.9%**
- **Function coverage – 18.6%**
- **Branch coverage – 9.4%**


LLVM Code Coverage Report

Directory: ./
Date: 2022-04-27 13:18:54
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%

	Exec	Total	Coverage
Lines:	41703	300655	13.9%
Functions:	24660	132931	18.6%
Branches:	19833	210451	9.4%

File	Lines			Functions		Branches	
include/llvm/ADT/APFixedPoint.h	🟥	66.0%	31 / 47	66.7%	16 / 24	72.7%	16 / 22
include/llvm/ADT/APFloat.h	🟨	88.6%	310 / 350	94.8%	110 / 116	65.9%	195 / 296
include/llvm/ADT/APInt.h	🟩	90.6%	529 / 584	88.2%	157 / 178	73.8%	186 / 252
include/llvm/ADT/APSInt.h	🟥	74.2%	72 / 97	76.3%	29 / 38	73.3%	44 / 60
include/llvm/ADT/AllocatorList.h	🟩	98.5%	66 / 67	80.3%	98 / 122	75.0%	3 / 4
include/llvm/ADT/Any.h	🟩	96.6%	28 / 29	50.0%	35 / 70	75.0%	3 / 4
include/llvm/ADT/ArrayRef.h	🟥	73.2%	120 / 164	11.1%	188 / 1694	40.0%	16 / 40
include/llvm/ADT/BitVector.h	🟩	98.5%	388 / 394	100.0%	93 / 93	79.9%	151 / 189
include/llvm/ADT/Bitfields.h	🟩	100.0%	37 / 37	46.4%	85 / 183	64.3%	9 / 14

Fig 4. LLVM Code Coverage Report

We can infer from the above figure that the Line coverage of this project is **13.9%**, function coverage is **18.6%** and branch coverage is **9.4%**. The branch, functional and line coverages are so less that it seems that many of the different possible conditions are not being tested i.e., the test cases generated by the testers are not testing the complete code written by developer, which can be possibly considered dead in tester's view. If there a scenario exists in real-time, where this dead code is executed, then there is good possibility of occurrence of bugs and errors.

If we analyze the state of testing for LLVM, it is evident that there are less tests written by the developers which might add to the low coverage. Even though the testers have written test cases accounting to **87%** of the total cases, the test coverage is significantly low. We cannot merely assume that this low coverage is just due to the poorly written test cases, but it could also be due to the inclusion of several config files, benchmark, binding or other project unrelated files, which are still accounted for in the coverage report. As there are minimal test cases, there is a possibility that wide range of bugs could not be identified lucidly. Even if some bugs are fixed, to test the cases, the entire project needs to be rebuilt, which takes up some more time. This could potentially affect both development and testing cycles. Due to the lower count of tests, if bugs arise during UAT or production, developers need to spend more resources on bugs than the actual development and testers need to do their sanity, smoke or regression suites again once the developer's code is ready, also the developed modules would not provide enough confidence to move to the production phase. This long-winded process could also delay the whole production process.

By analyzing the above points, we conclude that there are less adequate test cases to accommodate the high coverage.

Appropriateness of testing process

Frequency of Adding Tests

The below figure generated based on the stats given by the Pydriller analysis, represents the yearly test file generation count over the years **2001** to **2019**. We can observe that the frequency of test creation or modification has drastically increased from **2011** to **2013** and a similar trend is being observed in the following years.

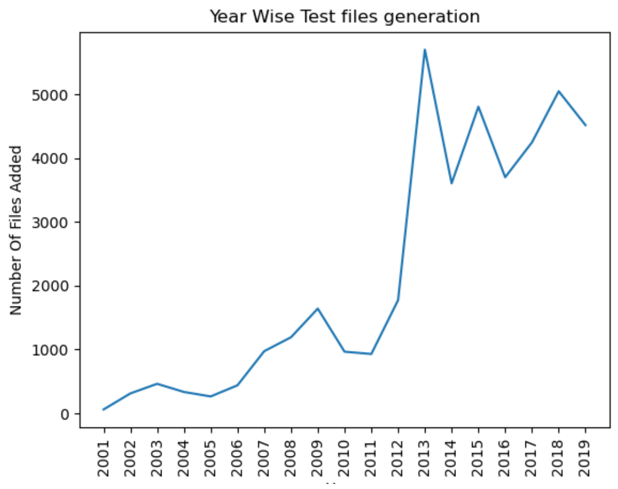


Fig 5. Year Wise Test Files Generation

On a thorough analysis of LLVM's GitHub, we observed that number of commits made to the regression folder over the years is so high when compared to the number of commits made to the unit test folder. It is also quite possible that the most stable and important test cases have been moved from unit test folder to the regression folder.

LLVM project is one of most active projects in GitHub, accounting to **300,000** plus commits to the entire project in the last three years. From the developer's perspective the project is moving at a rapid speed, but the number of unit tests being added are still low. As a tester, there are **10,000** new test cases that are added to the regression folder in the last **3 years**. If we see based on the coverage, there is still a huge gap between the number of commits made to the actual development and the number of commits made to the test folders. We believe, a greater number of tests should be included to reduce this gap and avoid any potential bugs for near future.

Time taken to run tests

We can consider time taken to test as a test appropriateness metric. The **regression tests** took about **1597.42s** i.e., **26 minutes** to run as opposed to their large number and **unit tests** took about **31.30s** to run. The build time for a serial build would take about **8 – 10 hours** and a parallel build would take about **4-5 hours**. For a project of this size, the obtained build time and test times are expected.

Test Stability Rate

We consider this metric to assess the stability and efficiency of test scripts to continuously function over a long period of time. As the false failures for this project accounts to only **0.33%** due to just **161** failed cases, we can conclude that this project has a high-test stability rate. This metric is ideally positive for both developers and testers for this project.

Test design efficiency

Considering the span of 22 years since the inception of LLVM project, on an average **~7** test cases are designed per day. This count is significantly low attributing to the huge number of developers contributing to this project over years. The pace of developing production files has no match with the test cases implementation.

Assert statements in Unit and Regression Test folders

From our analysis, the Regression test folder has **22712** test files, and the Unit test folder has **377** test files. Both the folders account for **77193** assert statements.

From the following plot, we can interpret that there are a greater number of assert statements in Regression Test folder compared to Unit Test folder. There are more asserts in regression suite, this is a good sign as a wide range of functional testing is performed each time a build is triggered. In Tester's/Business perspective, this ensures a good level of confidence when a new development is done by the developer.

As the developer's contribution for this project is invariably huge, the test cases or asserts for the developer's code i.e., the unit tests should be more. But this is not the case in this project, the assert statements are quite low, this can also account for the low path coverage and a lower confidence level.

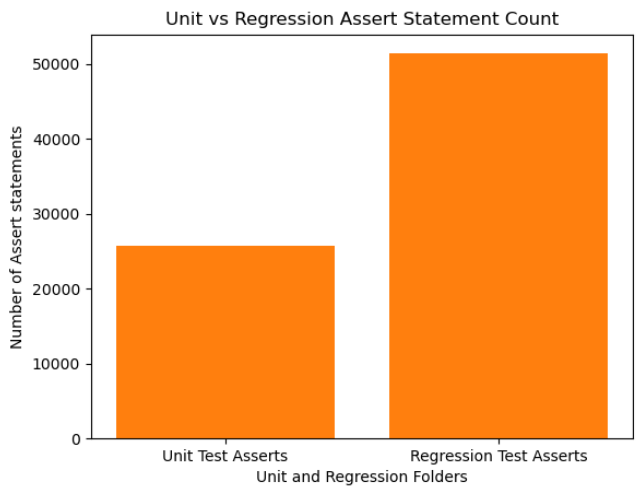
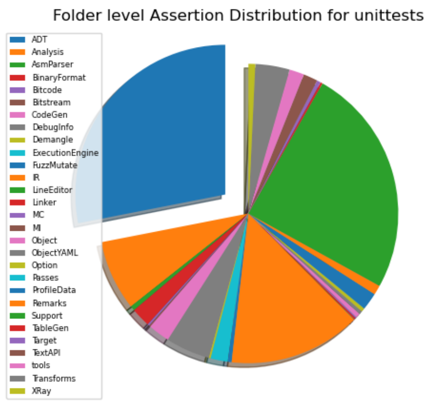


Fig 6. Assert statement count in unit test and regression test folders

The following figures depict the assert statements count in folder level for both **Unit** and **Regression** folders. The Abstract Data Type (**ADT**) sub-folder in unit tests folder has the highest assert count (**7234**) among the other sub-folders. The CodeGen sub-folder in Regression test folder has **32838** assert statements which is highest among the rest of sub-folders.



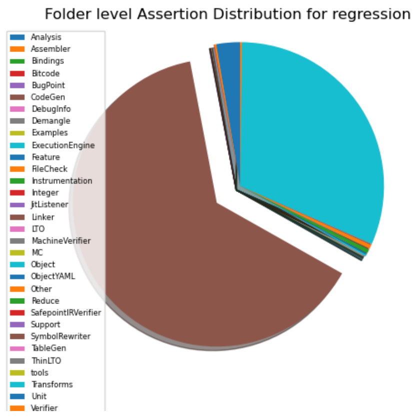


Fig 7 & 8. Folder level Assert statements

Debug and Assert statements in production files

The number of debug statements in the production files are summed up to **7982**. In developer's view, these statements will help in finding the root cause of the problem in the earlier stages which can resolve the errors and bugs fast. It is possible to troubleshoot an application in production without shutting it down. This is beneficial because other developers can parallelly work in fixing the bugs.

The number of assert statements in the production files are **19245**, each production file is densely populated with assert statements. This is surprising, attributed to the high performance of LLVM despite having a good number of asserts and debugs for the debug build.

Analysis of Test Files using Pydriller

The below plot illustrates the top 5 authors who made the greatest number of commits to the test folder for the year **2019**.

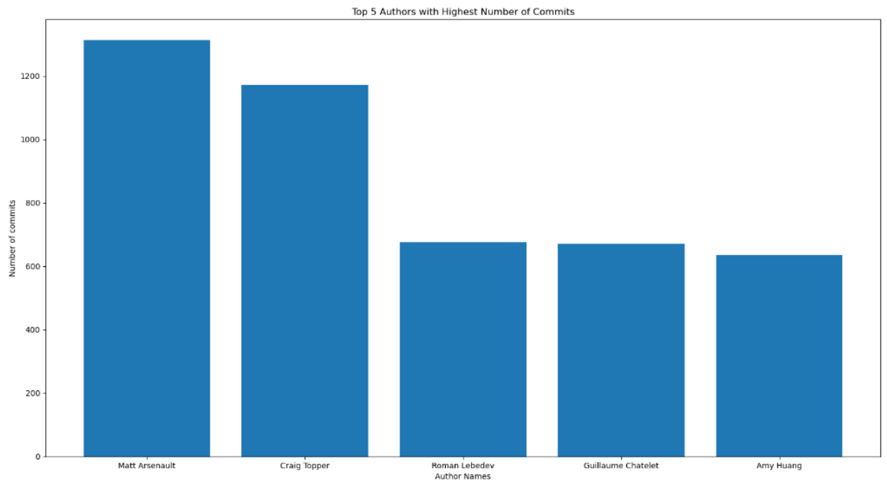


Fig 9. Top five authors with commits

From the Pydriller analysis for this project, we inferred that about **8059** test files have been newly created over the years **2001** to **2009**. There are about **78.6%** of test files modified frequently during this period, whereas the remaining **21.4%** remained un-modified which are related to type-notation and reference memory. From the tester's perspective, as development progressed, the number of test file commits increased as well, which is a good omen.

Around **24044** test files were newly generated over the next **10 years** i.e., **2010** to **2019**, and about **60.4%** of these files have been modified at least once, the rest **~40%** remained un-modified. We can observe that there is a steeper rise in test file generation for the last ten years than in the previous decade.

For a span of **2001** to **2019**, there are at least **20862** files that have been modified. There are a significant number of files that are getting modified every year, but, on an average, we can observe that each test file is being modified at least once every month.

Overall, there are **698** total contributors based on the Pydriller analysis. Out of these there are only **105** number of contributors to the test folders. This count for test contributors is quite low. The test coverage, confidence level and the robustness of the project could be potentially increased with the addition of more test files to the test folders.

Conclusion

From the above analysis, the existing tests have a success rate of **98.7%** which is a very good metric for quality assurance. However, we observed that the current state of testing is not satisfactory in terms of code coverage and the pace at which the test suite is being updated as opposed to the number of developers and testers. Based on the total number of contributors for this project, there are only **15%** of them involved in testing, which might be the main reason for having a smaller number of tests written that attributes to the low code coverage. This possibly cannot provide an acceptable level of confidence for developers, testers, and business. To provide more assurance, there should be more contribution from developers and testers to test folders which eventually leads to high code coverage.