Path based test coverage for quick test feedback

February 04, 2023
Agenda

- Introduction
- LLVM testing suite
- LLVM source-based code coverage
- Implementation
- Demo
Introduction

Most of the day to day tests in LLVM are regression tests executed by Lit, structured as source code or IR to be passed to some binary, rather than test code directly calling the code to be tested.

The goal of this work was to help developers create good test coverage for their patch and enable reviewers to verify that they have done so.

To accomplish this we have introduced a tool that can be fed a patch as input, add coverage instrumentation for the affected source files, runs Lit tests, and records which test cases cause each counter of source code to be executed.
Regression tests

The regression tests are small pieces of code that test a specific feature of LLVM or trigger a specific bug in LLVM. The language they are written in depends on the part of LLVM being tested. These tests are driven by the Lit testing tool (which is part of LLVM), and are located in the llvm/test directory.

Unit tests

Unit tests are written using Google Test and Google Mock and are located in the llvm/unittests directory. In general unit tests are reserved for targeting the support library and other generic data structure, we prefer relying on regression tests for testing transformations and analysis on the IR.
Unit test

```cpp
    1     /**
    2        * Unit test for the function.
    3        *
    4        * @return true on success.
    5     */
    6     static bool test()
    7     {
    8        // Test case 1
    9        // Check if the function returns the expected result.
    10       if (test() != expected_result)
    11       return false;
    12
    13       // Test case 2
    14       // Check if the function returns the expected result.
    15       if (test() != expected_result)
    16       return false;
    17       return true;
    18    }
```
LLVM source-based code coverage

The code coverage workflow consists of three main steps:

Compiling with coverage enabled.
clang++ -fprofile-instr-generate -fcoverage-mapping foo.cc -o foo

Running the instrumented program.
LLVM_PROFILE_FILE="foo.profraw" ./foo

Creating coverage reports.
llvm-profdata merge -sparse foo.profraw -o foo.profdata
llvm-cov show ./foo -instr-profile=foo.profdata
Sample coverage report

```c
#include <stdio.h>

int main() {
    int num;

    // Input a number
    printf("Enter an integer: ");
    scanf("%d", &num);

    // Check if the number is even or odd using if-else loop
    if (num % 2 == 0) {
        printf("%d is an even number.\n", num);
    } else {
        printf("%d is an odd number.\n", num);
    }

    return 0;
}
```
Implementation

https://reviews.llvm.org/D154280 ([LIT] Added an option to llvm-lit to emit the necessary test coverage data, divided per test case)

https://github.com/llvm/llvm-project/pull/71841 ([Code Coverage] Add a tool to check test coverage of a patch)
Functions implemented for tool

1. configure_logging
2. custom_print
3. create_patch_from_last_commit
4. extract_source_files_from_patch
5. write_source_file_allowlist
6. extract_modified_source_lines_from_patch
7. build_llvm
8. run_single_test_with_coverage
9. run_modified_lit_tests
10. run_modified_unit_tests
11. process_coverage_data
12. read_coverage_file_and_return_result
13. print_coverage_details
14. print_common_uncovered_lines
15. report_covered_and_uncovered_lines
16. parse_suite_info
17. find_lit_tests
18. parse_args
19. main
GitHub CI workflow
Demo

https://github.com/llvm/llvm-project/pull/71894 ([Clang] add user-level sizeless attribute)

https://github.com/xgupta/llvm-project/actions/runs/6994512086/job/19028363714#step:5:3726

https://github.com/xgupta/llvm-project/actions/runs/6994498992/job/19028333944#step:5:2609

https://github.com/llvm/llvm-project/pull/72273 ([AArch64] Add an AArch64 pass for loop idiom transformations)
Result
Questions?

Write to me on shivam98.tkg@gmail.com

or connect

![QR Code](qr-code-url)