Aerodrome_Analysis

Goutam Das, Nikhil Meena

21 April 2025

1 Project Overview

The objective of this project is to implement **on-the-fly AeroDrome analysis** using dynamic instrumentation techniques. Specifically, we aim to detect atomicity violations in multithreaded programs by applying the AeroDrome algorithm, which performs atomicity checking in linear time using vector clocks.

We opted to use **LLVM** for this purpose, which provided source-level instrumentation.

Our approach involved two key stages:

1. **Instrumentation using LLVM**: We instrumented the source code to log relevant memory accesses and synchronization events during execution.
2. **AeroDrome Analysis on Execution Trace**: We applied the AeroDrome algorithm on the generated execution trace to efficiently detect atomicity violations and identify conflicting accesses.

2 LLVM Setup

We have used **LLVM version 18** for this project.

To install a specific version of LLVM on a Debian-based system, we followed the steps below:

```
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 18
```

This script automatically sets up the required repository and installs the chosen LLVM version along with related tools and dependencies.

3 Project Set-up and Run

The github repo link for the project is :

`https://github.com/Akhilstaar/Aerodrome-analysis`

The complete code and required files are hosted on GitHub. You can clone the repository using the following command:

```
git clone https://github.com/Akhilstaar/Aerodrome-analysis
```

The project has the following folder structure:

- **Analysis/**
  Contains all the necessary source files and scripts to perform the AeroDrome analysis on the generated trace logs.
- **Instrumentation/**
  Contains the LLVM-based instrumentation logic to instrument multithreaded programs and generate memory access traces.
- **Testcases/**
  A collection of C++ test files that demonstrate various multithreaded scenarios. These are the input programs to be instrumented and analyzed.

There are three main scripts in the root directory to assist with automation:

- `maketrace.sh`
  Takes one argument — the name of a file from the `Testcases/` folder. This script compiles and instruments the file, then runs it to generate `trace.log` in the working directory.
  Example command:

  ```
  chmod +x maketrace.sh
  ./maketrace.sh t1.cpp
  ```

- `analyze.sh`
  Requires no arguments. This script performs the AeroDrome analysis on the existing `trace.log` file in the working directory and outputs the result.
  Example command:

  ```
  chmod +x analyze.sh
  ./analyze.sh
  ```

- `Runall.sh`
  Runs all the test cases present in the `Testcases/` directory one by one, generating traces and performing AeroDrome analysis on each. The outputs are displayed sequentially.
  Example command:

  ```
  chmod +x runall.sh
  ./runall.sh
  ```

4 Testing and Results

To evaluate the correctness of our AeroDrome-based data race detection tool, we constructed a set of test programs. These programs are designed to cover a variety of thread interleaving patterns that typically result in data races. For each concurrency pattern, we implemented both:

- a *racy version* (with unsynchronized accesses), and
- a *non-racy version* (with proper synchronization mechanisms).

Test Coverage

We tested the tool over the following 7 representative interleaving patterns:

- **WWW (Write-Write-Write)**
  `t2_www_racy.cpp`, `t2_www_not_racy.cpp`
- **RRW (Read-Read-Write)**
  `t3_rrw_racy.cpp`, `t3_rrw_not_racy.cpp`
- **RWW (Read-Write-Write)**
  `t4_rww_racy.cpp`, `t4_rww_not_racy.cpp`
- **WWR (Write-Write-Read)**
  `t5_wwr_racy.cpp`, `t5_wwr_not_racy.cpp`
- **WRR (Write-Read-Read)**
  `t6_wrr_racy.cpp`, `t6_wrr_not_racy.cpp`
- **RRR (Read-Read-Read)**
  `t7_rrr_racy.cpp`, `t7_rrr_not_racy.cpp`
- **RWR (Read-Write-Read)**
  `t8_rwr_racy.cpp`, `t8_rwr_not_racy.cpp`

In addition to these structured tests, we also included a general-purpose concurrency test program `t1.cpp` to further validate the tool under less predictable concurrent access scenarios.

Representative Result: `t6_wrr_racy.cpp`

As an example, in the WRR pattern test `t6_wrr_racy.cpp`, thread 2 performs an unsynchronized write while thread 3 performs reads on the same memory address. Our tool successfully detects the race condition:

```
=== DATA RACE DETECTED ===
  Operation:    WRITE
  Thread:       2
  Log Line No.: 39
```

```
Address:      0x6151143a210c
Conflicting with threads: [3]
Transaction begin clock: [2, 1, 2, 0, 0]
Resource clock:          [2, 1, 2, 2, 0]
```

Summary

The results of our test suite strongly support the correctness of our detection logic:

- *All racy programs* were correctly flagged with precise and detailed conflict information.
- *All non-racy programs* executed without false positives.

5 Improvement Scopes

Here are some improvement ideas for our current project :

- Improving efficiency of Aerodrome Analysis (not asymptocially but constant factor increase by optimizing read clocks)
- better transaction handling (currently transaction granularity is a function or module level)
- Analysis for nested transactions
- Integrating Aerodrome Analysis in LLVM pass

6 References

- LLVM Documentation: `https://llvm.org/docs/`
- LLVM Tutorial: `https://llvm.org/docs/tutorial/`
- AeroDrome Paper: Atomicity Checking in Linear Time using Vector Clocks
- Stack Exchange and Stack Overflow discussions related to LLVM and dynamic instrumentation were referred to during debugging and implementation.