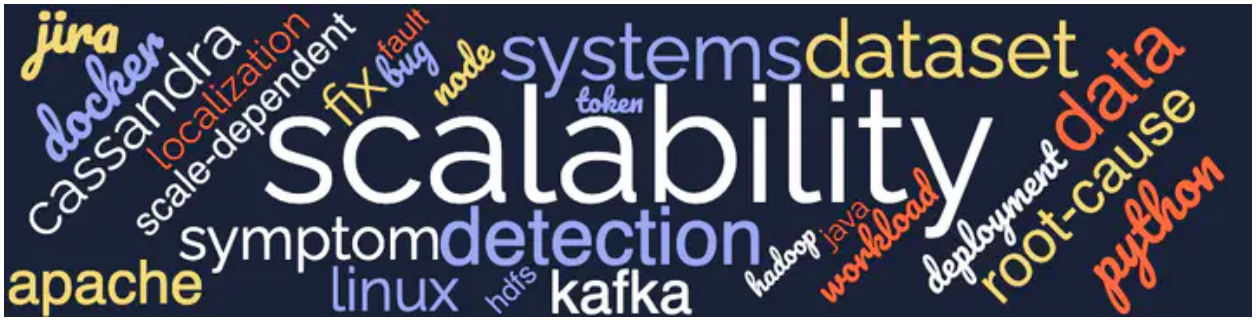
ZAHRA NABILA MAHARANI



Google Summer of Code 2023 - ScaleBugs: Reproducible Scalability Bugs

Project Description

- **Topics**: Scalability systems, bug patterns, reproducibility, bug dataset
- **Skills**: Linux Shell, Docker, Java, Python
- **Difficulty**: Medium
- **Size**: Large (350 hours)
- **Mentors**: Cindy Rubio González, Haryadi S. Gunawi, Hao-Nan Zhu
- **Contributor(s)**: Goodness Ayinmode, Zahra Nabila Maharani

Introduction

Our work has been on packaging buggy and fixed versions of scalability systems, a runtime environment that ensures reproducibility, and the workloads used to trigger the symptoms of the bug inside docker containers. By packaging these versions together, we are simplifying the process of deployment and testing. This enables us to switch between different versions efficiently, aiding in the identification and comparison of the bug's behavior. For each scalability bug, we have carefully built a runtime environment that is consistent and reproducible. This approach ensures that each time we run tests or investigations, the conditions remain identical.

ScaleBugs: Reproducible Scalability Bugs | UCSC OSPO

Significance of our work

Our project's significance is rooted in building a dataset of buggy and fixed versions of distributed systems within Docker containers. This dataset forms a vital component of a larger framework called 'ScaleView,' which is a framework for identifying and analyzing potential scalability faults in large-scale distributed systems. The inclusion of known buggy and fixed versions within the dataset serves as a valuable asset towards providing a solid foundation for testing and enhancing bug-finding methodologies and other experiments.

Current state.

Unfortunately at this time the repo for our project is private because the ScaleView framework is still in development. But the docker images we created have been pushed to docker hub and could be viewed through this link.

DockerHub
https://hub.docker.com/r/ucdavisplse/scalability-bugs/tags

How to Reproduce the Bugs

Step 0

Build the Docker image if you have no access to the Docker Hub Repo.
i.e. docker build . -t ucdavisplse/scalability-bugs:HD14366

Command
docker build . -t ucdavisplse/scalability-bugs:[Bug]

Step 1
Reproduce the bug with the buggy version of Cassandra.

Command

bash reproduce_buggy.sh

Step 2

Reproduce the bug with the fixed version of Cassandra.

Command

bash reproduce_buggy.sh

Step 3

Compare the execution time of the two versions, done.

(Link to our commit and PRs when ScaleView becomes public)

Commits

- **Goodness's Commits**
- **Zahra's Commits**

Pull Request

- **Goodness's PR**
- **Zahra's PR**

Accomplishments

We have successfully built a dataset of reproducible scalability bugs. Each individual bug artifact within the dataset comprises the following key components:

1. **Bug Description**: Detailed description of the bug's characteristics and root causes of the bug.
2. **Version Information:** Identification of both the buggy and fixed versions of the scalability system.
3. **Runtime Environment:** Creation of a runtime environment ensuring accurate reproducibility of the bug.
4. **Workload Shell Script:** Development of a specialized script that effectively showcases the bug's symptoms across varying scales.
5. **Dockerfile:** Inclusion of Dockerfiles for simplified the process of bug reproduction.

IGNITE 12087

This bug stems from the resolution of the IGNITE-5227 issue (another bug), which has led to a significant decline in the performance of a particular operation. Prior to addressing IGNITE-5227, the insertion of 30,000 entries displayed remarkable efficiency, completing in roughly 1 second. However, post the resolution, executing the same insertion process for 30,000 entries witnessed a considerable slowdown, taking approximately 130 seconds – a performance degradation of nearly 100 times.

CASSANDRA 14660

This bug is related to how clusters work together and how a lock is causing conflicts with the critical path. The issue arises from a method call that uses O(Peers * Tokens) resources while contending for a lock, which is causing problems in the write path. The lock is used to protect cached tokens that are essential for determining the correct replicas. The lock is implemented as a synchronized block in the TokenMetadata class.

How was this fixed?

It was fixed by reducing the complexity of the operation to O(Peers) taking advantage of some properties of the token list and the data structure.

CASSANDRA 12281

This bug is also related to how clusters work together and a lock conflict. The issue arises when a specific method is trying to access a lot of resources (O(Tokens^2)) while contending for a read lock. As reported, a cluster with around 300 nodes has around 300 * 256 (assuming the default number of tokens) tokens, thus joining a new member reportedly is taking more than 30 mins. This happens because due to the long execution time here, this lock makes every gossip message delayed, so the node never becomes active.

How was this fixed?

The granularity of the lock is decreased, meaning that the expensive function calls now do not take the problematic read lock and simply use a synchronized block, synchronizing on a specific field, that does the job much better.

HA16850

This is a bug related to obtaining thread information in the JvmMetrics package. When obtaining thread information, the original buggy version used MXBeans to obtain thread information. The call uses an underlying native implementation that holds a lock on threads, preventing thread termination or creation. This means that the more threads that we have to obtain information for, the longer the function call will hold a lock. The result is that the execution time scales on the number of active threads O(threads).

How was this fixed?

Developers utilized a ThreadGroup to keep track of obtaining metrics for threads. The result is that there is no lock held for every thread.

CA13923

This issue revolves around conflicts between the “flush” and “writes” processes. The main problem is that during the “flush” process, a resource-intensive function called

“getAddressRanges” is invoked. This function has a high computational cost and its complexity is O(Tokens^2). In other words, the time it takes to complete this function grows quickly as the number of “tokens” increases. This situation is causing challenges and delays in the overall process.

How was this fixed?

This function call affected many paths and they made sure no one calls getAddressRanges in critical paths.

Any challenges or important things you learned during the project.

Demanding Memory Requirements: Running certain builds consumes a significant amount of memory. This places a strain on system resources and can impact the overall performance and stability of the process.

Little Issues Impacting Execution: Often, seemingly minor details can obstruct the successful execution of a build. Resolving such issues requires thorough investigation and extensive research into similar problems faced by others in the past.

Complexities of Scalability Bugs: Identifying the underlying causes of scalability-related bugs is intricate. These bugs exhibit unique characteristics that can complicate the process of pinpointing and comprehending their root origins.

Future Work (What's left to do ?)

Looking ahead, ongoing efforts will be centered around the continued population of the scalebugs dataset with Docker containers of buggy and fixed versions of various distributed systems.

Acknowledgments

Thank you GSoC for your support during this project. We also want to thank OSRE, who provided the conducive environment for mentorship and guidance, which has been invaluable

in shaping our work. Our deepest thanks go to our mentors for their exceptional assistance, support, and patience. Their insights have been pivotal in our growth and progress.