Indonesian Journal of Electrical Engineering and Computer Science
Vol. 40, No. 2, November 2025, pp. 789~800
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v40.i2.pp789-800

Experimental analysis and bug abstraction for distributed computation on ray framework

Arnaldo Marulitua Sinaga, Wordyka Yehezkiel Nainggolan
Department of Applied Science in Software Engineering Technology, Institut Teknologi Del, North Sumatra, Indonesia

Article Info

Article history:

Received Oct 30, 2024
Revised Aug 27, 2025
Accepted Oct 14, 2025

Keywords:

Bug
Bug abstraction
Distributed computing
Experimental analysis
Ray framework

ABSTRACT

This research aims to address challenges in distributed computing, focusing on the ray framework, which has potential for efficient parallel and distributed task execution. While methods such as model-checkers and fuzzing have been applied to detect bugs, both have limitations in handling the complexity of distributed computing, particularly in dealing with issues like state-space explosion and identifying rare bugs. This study proposes an alternative approach through experimental analysis and bug abstraction methods to discover, identify, and classify bugs in the ray framework. Experimental analysis involves isolating and re-testing bugs in a controlled environment to understand their characteristics, while bug abstraction analyzes the factors causing bugs to identify common patterns and characteristics. The results of this research successfully identified three main categories of bugs: crash, performance, and inaccurate status, and revealed bug characteristics that do not depend on actor instance multiplicity, actor type, specific event sequences, or particular configurations. This research makes a significant contribution to the development of more effective and efficient bug detection methods in distributed computing, particularly in the ray framework, and paves the way for further research to enhance the reliability of distributed systems.

Corresponding Author:

Wordyka Yehezkiel Nainggolan
Department of Applied Science in Software Engineering Technology, Institut Teknologi Del
North Sumatra, Indonesia
Email: if420041@students.del.ac.id

1. INTRODUCTION

In the rapidly evolving digital era, distributed computing plays a crucial role in connecting numerous computers and large-scale internet infrastructures, facilitating collaboration and communication among computers or devices [1]. This technology enables computational processes to be distributed across multiple locations rather than being confined to a single centralized system. Distributed computing is a concept in which computing components are spread across different locations instead of being concentrated in one place. These components work simultaneously to solve complex tasks, breaking down large workloads into smaller ones that can be executed in parallel [2]. This parallelism significantly improves efficiency and processing speed [3]. Moreover, distributed computing enhances system reliability, as failures in one component can be mitigated by other components taking over the workload, thereby minimizing downtime.

As data processing techniques continue to evolve, distributed computing has become a widely adopted and essential method for handling large-scale computational tasks. However, several key challenges in distributed computing have been identified, including compatibility issues, domain constraints, heterogeneity, and security concerns [2]. Addressing these challenges is crucial to optimizing the performance and robustness of distributed systems. One of the significant challenges in distributed

computing is the straggler effect, where communication and computation processes among nodes become unsynchronized, leading to performance degradation. Research by Sun *et al*. titled "Coded computation across shared heterogeneous workers with communication delay" explores methods to mitigate this effect by improving efficiency in distributed computing [4]. Ensuring synchronization among distributed nodes remains a critical factor in enhancing system performance.

Another fundamental issue in distributed computing is the presence of bugs, which refer to errors or malfunctions in software programs [5]. Research in TaxDC categorizes bugs in distributed systems based on non-deterministic concurrency (DC) errors. This study analyzed 104 DC-related bugs across four large-scale distributed data processing systems: Cassandra, Hadoop MapReduce, HBase, and ZooKeeper [6]. Similarly, the network error analysis tool (NEAT) study focuses on network partition failures in cloud systems. When a network partition occurs, devices within the affected network lose communication, causing disruptions. Their study documented 136 system failures due to network partition errors across 25 distributed systems [7]. Lastly, the Agamotto study classifies bugs in persistent memory (PM) applications, a type of memory that retains data even after power loss, eliminating the need for file systems. This research identifies two primary categories of bugs: missing bug flush/fence and extra bug flush/fence, both of which impact data consistency and reliability in PM systems [8].

To address these challenges, many researchers have proposed solutions such as using model-checker methods. A model-checker is a tool used to verify whether a system meets specific requirements by exhaustively exploring all possible states. However, when applied to large workloads, this method encounters a problem known as state-space explosion, where the number of potential system states becomes unmanageably large [9], [10]. Additionally, automated testing techniques like fuzzing have been employed to detect bugs. Fuzzing involves injecting large amounts of random or unexpected input data into a system to uncover errors that might not be detected through conventional testing. However, this approach struggles with identifying rare bugs, as most inputs fail to trigger meaningful errors, making the process inefficient [11]. Consequently, there is a pressing need for more effective and efficient methods for identifying and addressing bugs in distributed computing systems.

Given these challenges, this research applies experimental analysis and bug abstraction methods to detect and categorize bugs in the ray framework. Ray is a distributed computing framework designed for efficient execution of parallel and distributed tasks [12]. Unlike model-checkers and fuzzing, which have limitations in handling complex distributed environments, our approach focuses on systematically re-examining previously observed bugs to understand their characteristics and categorize them accordingly. Experimental analysis involves executing and isolating bug occurrences in a controlled environment to investigate their root causes, while bug abstraction systematically analyzes contributing factors to identify recurring patterns in bug behavior.

The choice of ray as the testing platform is motivated by its capabilities in executing distributed tasks across multiple locations while efficiently handling large-scale data processing [12]. Ray's architecture consists of three primary layers: Ray AI Libraries, Ray Core, and Ray cloud. This research focuses on the Ray Core layer, which enables developers to build scalable Python applications and accelerate machine learning workloads. The Ray Core layer is responsible for task distribution, scheduling, and inter-node communication, making it crucial for achieving high performance and scalability in distributed computing [13]. Other distributed computing frameworks, such as Hadoop, Spark, and Storm, offer distinct advantages in different scenarios. Hadoop is optimized for offline batch data processing but is relatively slow. Spark provides faster large-scale data processing, making it preferable for real-time analytics. Meanwhile, Storm specializes in real-time stream processing, enabling efficient handling of continuous data flows [14]. Each of these frameworks serves specific use cases, but research on bug detection in ray remains limited, highlighting the need for further investigation.

The strengths and weaknesses of the ray framework have been analyzed in studies such as "Ray-based Elastic Distributed Data Parallel Framework with Distributed Data Cache" by Lin *et al*. and "Boost the Performance of Model Training with the Ray Framework for Emerging AI Applications" by Ruan *et al*. [14], [15]. The advantages of ray framework include fault tolerance (Ray can recover from failures without significant data loss), scalability (users can adjust the number of training processes dynamically), high performance (optimized for high-performance distributed execution), and flexibility (supports various programming models). However, ray also has setup complexity (users must understand multiple concepts and configurations) and resource management challenges (proper resource allocation is necessary to avoid bottlenecks).

The decision to focus on the ray framework for testing stems from the limited research conducted to analyze bugs in this computing framework. Previous studies highlight ray's advantages in increasing efficiency. For example, research by Lin *et al*. demonstrated that Ray improved data processing efficiency by up to twice the speed of PyTorch's dataloader on a 10-Gigabit Ethernet cluster [15]. Similarly, Ruan *et al*.

found that ray reduced model training time by up to 50% without compromising accuracy [16]. Additionally, Sheikh *et al*. showed that ray accelerated training for knowledge graph embedding models by a factor of twelve while maintaining evaluation metrics [17]. These studies emphasize ray's performance benefits, but none focus on systematically identifying and analyzing its software bugs.

Experimental research is a well-established method for examining cause-and-effect relationships, including scientific and engineering challenges. According to Sylwester *et al*. in "Experimental Design and Biometric Research: Toward Innovations", experimentation plays a crucial role in addressing contemporary scientific issues, including sustainable development, through well-defined problem statements and causal analysis [18]. In distributed computing, experimental research often requires simulations or controlled test environments. For instance, McKevett applied brief experimental analysis (BEA) to diagnose learning difficulties in mathematics and match interventions to specific student needs [19]. Similarly, Mellott used BEA to improve students' multiplication skills, demonstrating its effectiveness in tailoring solutions to specific problems [20]. These examples underscore the relevance of experimental analysis in identifying and mitigating issues in distributed computing systems. Furthermore, abstraction is a fundamental principle in computational research. According to Beren Millidge in "Towards a Mathematical Theory of Abstraction", abstraction involves creating simplified representations of complex systems that retain essential characteristics while omitting irrelevant details. This process enables researchers to focus on key aspects, improving analytical efficiency and decision-making [21].

In conclusion, this research differs from previous studies by applying experimental analysis and bug abstraction to systematically identify and categorize bugs within the ray framework. While prior research focused on improving ray's performance and efficiency, this study aims to deepen the understanding of its software bugs, ultimately contributing to the enhancement of distributed computing reliability.

2. RESEARCH METHOD

The research method, as described in Figure 1, is structured into three key stages, such as preparation, experimental analysis, and bug abstraction. Each stage plays a vital role in the overall research process, beginning with the preparation phase, which ensures a solid foundation for the experiments. The details of each process are explained in the following subsections.

2.1. Preparation

This phase involves several critical steps to lay a solid foundation for the research, ensuring that the experimental process is thorough and consistent. The details of these steps are as follows,

2.1.1. Collect bug dataset

The collection of the bug dataset for the ray framework involves a multi-step process. Initially, researchers access the public GitHub repository of the ray framework. Within this repository, the "Issues" section is selected to identify relevant bug reports. Issues are filtered to include numbers from #20000 to #40000, targeting reports from late 2021 or early 2022 up to the end of 2023 to ensure relevance and recency. The dataset is further refined using filters, such as "is:closed" to select only resolved issues, "is:bug" to focus on actual bugs rather than feature requests or documentation updates, and "is:core" to ensure that the issues pertain to the core components of the ray framework. Issues are then examined for those with associated pull requests and reproduction scripts, as these provide valuable insights into the components causing the bugs and the conditions under which they occur.

2.1.2. Build testing environment

The creation of the testing environment, as shown in Algorithm 1, is executed through Docker, which provides a containerized setup for isolating and managing dependencies. The process begins by selecting a suitable base Linux image for the Docker container. The system is then updated, and Git is installed to facilitate the cloning of the ray framework repository. Once the repository is cloned, the working directory is set to the project folder, and a specific commit related to the bug is checked out. Further system updates are performed, and essential packages such as Python3, ca-certificates, and others are installed. A virtual Python environment is created and activated to segregate Python dependencies from the main system. Additional libraries and tools, including Node.js, Bazel, and various build essentials, are installed. The final steps involve copying the necessary bug reproduction scripts into the container and configuring the entry point to bash, which provides an interactive shell when the container is running. This setup ensures that the testing environment accurately mirrors the conditions needed to reproduce and analyze the bugs.

Algorithm 1. Testing docker enviroment

```
FROM ubuntu:22.04

RUN apt-get update && \
 apt-get upgrade -y && \
 apt-get install -y git

RUN git clone https://github.com/ray-project/ray.git

WORKDIR ray

# checkout at parent commit version bug and set to BUG_VERSION value
RUN git checkout BUG_VERSION

RUN apt-get update && apt-get install -y \
 python3 \
 python3-pip \
 ca-certificates \
 nano

RUN apt install -y python3.10-venv

RUN echo 'alias python="python3"' >> ~/.bashrc
RUN echo 'alias pip="pip3"' >> ~/.bashrc

SHELL ["/bin/bash", "-c"]
RUN python3 -m venv venv

ENV VIRTUAL_ENV=venv
ENV PATH="$VIRTUAL_ENV/bin:$PATH"

#RUN pip3 install -U https://s3-us-west-2.amazonaws.com/ray-wheels/latest/ray-3.0.0.dev0-
cp310-cp310-manylinux2014_x86_64.whl

RUN apt-get install -y software-properties-common
RUN add-apt-repository -y ppa:ubuntu-toolchain-r/test
RUN apt-get install -y build-essential curl gcc-9 g++-9 pkg-config psmisc unzip
RUN update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 90 \
 --slave /usr/bin/g++ g++ /usr/bin/g++-9 \
 --slave /usr/bin/gcov gcov /usr/bin/gcov-9

RUN echo "insecure" >> ~/.curlrc

RUN source venv/bin/activate && ci/env/install-bazel.sh

RUN curl --silent -o- https://raw.githubusercontent.com/creationix/nvm/v0.39.0/install.sh
| bash
ENV NODE_VERSION=14.21.3
ENV NVM_DIR=/root/.nvm
RUN . "$NVM_DIR/nvm.sh" && nvm install ${NODE_VERSION}
RUN . "$NVM_DIR/nvm.sh" && nvm use v${NODE_VERSION}
RUN . "$NVM_DIR/nvm.sh" && nvm alias default v${NODE_VERSION}
ENV PATH="/root/.nvm/versions/node/v${NODE_VERSION}/bin/:${PATH}"
ENV NODE_PATH="$NVM_DIR/v$NODE_VERSION/lib/node_modules:${NODE_PATH}"
WORKDIR dashboard/client
RUN npm ci
RUN npm run build


WORKDIR ../../python
RUN source ../venv/bin/activate && pip3 install -e . --verbose

WORKDIR ..

RUN apt-get install -y vim

RUN pip install pytest
RUN pip install psutil
RUN pip install six

COPY ./script.sh /
COPY ./recreate.py /

ENTRYPOINT /bin/bash
```

2.1.3. Create reproduction and shell scripts

In this step, the focus is on developing scripts necessary for bug reproduction and testing. Researchers start by accessing one of the selected issues from the bug dataset. A reproduction script, named recreate.py, is created based on the details provided in the issue description. This script is designed to replicate the bug under controlled conditions, ensuring that the experiment mirrors the issues reported. Alongside this, a shell script named script.sh is developed to facilitate the execution of the reproduction script as outlined in Algorithm 2. This shell script automates the process of activating the Python virtual environment and running the reproduction script within a Unix/Linux environment, thereby streamlining the testing procedure and ensuring consistent execution across different trials.

Algorithm 2. Shell Script

```
source /ray/venv/bin/activate && python recreate.py
```

2.1.4. Build docker image and container

The construction of the Docker image and container involves several detailed steps. First, a Docker image is created using a Dockerfile, which specifies the necessary components, such as code, runtime, libraries, and dependencies, required for the application. The command "docker build -t image_name." is utilized to build the image, where "-t image_name" tags the image with a name, and. indicates that Docker should look for the Dockerfile in the current directory. After successfully building the image, the next step is to create and run a container from this image using the command "docker run -it --name container_name image_name". This command initiates a new container in interactive mode, allowing users to interact with the container through the terminal. The "-it" option ensures the container runs interactively, with "--name container_name" assigning a specific name to the container, and "image_name" refers to the Docker image used for container creation. This process ensures that the application runs in a consistent and isolated environment, facilitating accurate testing and analysis.

2.2. Experimental analysis

The experimental analysis phase is crucial for identifying and understanding the bugs within the system. It involves several steps bug discovery, identification, and isolation. The initial step involves running the shell script, which activates the testing environment, initiates the reproduction script, and ensures all necessary dependencies are present. This script helps maintain a consistent testing environment and verifies that all components are functioning as expected. Following the script execution, the log output is reviewed. This output contains detailed information about the execution process, including error messages and warnings, which are essential for detecting the presence of bugs. The next step is to compare the actual results obtained from the log with the expected outcomes. Discrepancies between these results indicate the presence of a bug. A thorough analysis is then conducted to identify the specific components within the system responsible for the bug, involving code examination and understanding the interactions between various system components. Once the bug is identified, the next step is to isolate it by examining the symptoms and conducting a root cause analysis. This process involves a detailed investigation of the observed symptoms and identifying the specific conditions that trigger the bug. By isolating the bug, researchers can determine its category and develop strategies for addressing and fixing it, thereby enhancing the overall reliability and performance of the system.

2.3. Bug abstraction

Following the application of experimental analysis, the bug abstraction method is employed to categorize bugs based on their key characteristics. This method leverages the findings from experimental analysis to systematically extract these characteristics. The abstraction process involves a series of targeted questions designed to illuminate the bug's core attributes. By systematically addressing each question, a comprehensive understanding of the bug's characteristic is established. The specific questions employed in this process are explained in detail below,

- Do we need multiple instances of a certain actor for the bug to happen?
  This question explores whether the bug occurs only when there are multiple instances of a certain actor operating simultaneously. Where, instances of an actor means various copies or objects of the actor running the same task. This helps in identifying whether the bug is related to concurrency or resource management issues.
- Do we need multiple types of actors for the bug to happen?
  This question focuses on the need to determine how many and different types of actors are involved when a bug arises. It is important to identify whether the bug is related to specific interactions between components or tasks in the system.
- Does it happen after a failure, or can it happen without a failure?

This question asks whether the bug arises as a direct result of a failure of the system or another component, or whether the bug can occur independently. It is important to understand whether the bug is reactive to a specific failure condition or can occur without a prior failure.

– Does it require a specific order of events to occur?
  This question aims to find out if the bug only appears when there is a specific order of events that must occur, which is not always guaranteed to happen. This suggests that the bug may be related to concurrency and execution order issues in the system. By understanding whether a specific sequence is required, it can help in identifying bugs caused by race conditions.
– Does it happen only for a specific configuration, or can it happen with any configuration?
  This question aims to find out if the bug only appears in a specific system configuration or can occur in various configurations. This is important to determine whether the bug is an environment-specific problem or a broader problem.

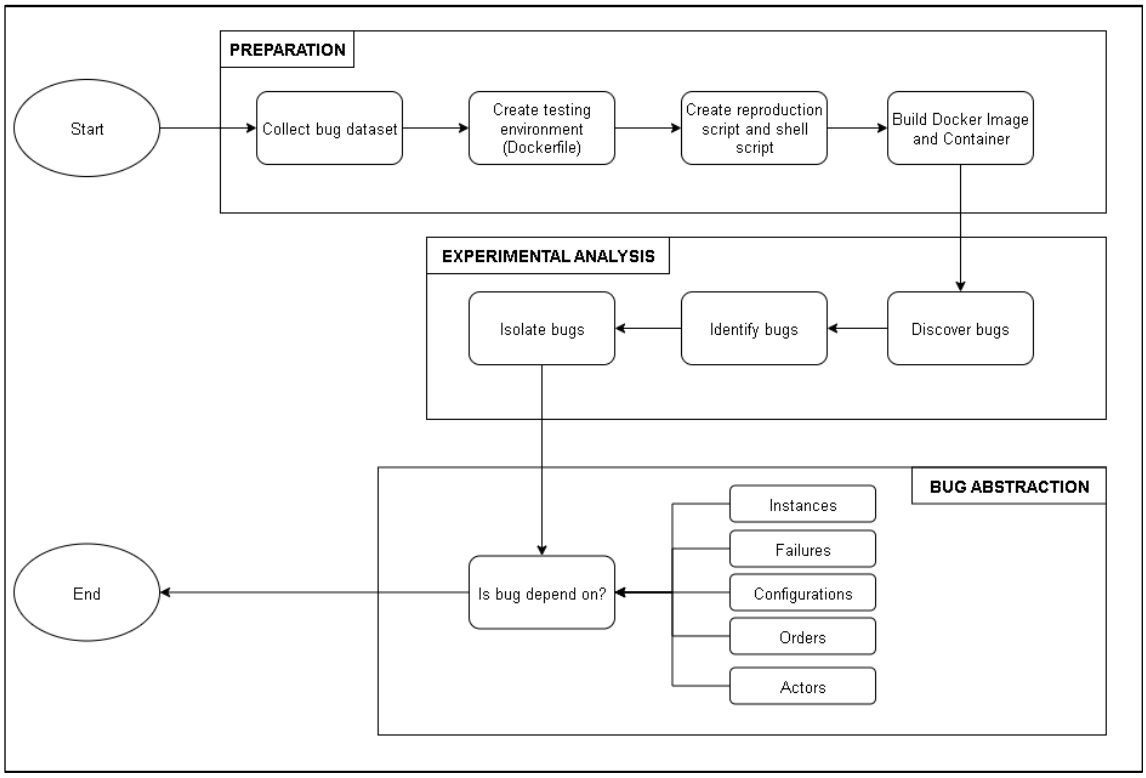

Figure 1. Main research procedure

3. RESULTS AND DISCUSSION

This section presents the results of the study and provides a discussion of the findings obtained through the implementation of the experimental analysis and bug abstraction methods. The discussion is structured to offer a comprehensive interpretation of the findings, comparisons with prior research, and implications for future studies.

3.1. Bug symptom categorization

The process of collecting bug datasets using the defined bug dataset collection method resulted in a dataset comprising 91 bugs. The experimental setup, which included test environment creation, reproduction and shell scripts, Docker image and container deployment, and the application of the experimental analysis and bug abstraction methods, allowed for the classification of bug symptoms and the determination of their frequency. Figure 2 illustrates the distribution of bug occurrences by category. The histogram presents the number of occurrences of each bug category, with the horizontal axis representing frequency and the vertical axis listing different bug categories. Each bar corresponds to the number of cases recorded for a given category. The following is an explanation of the bug categorization results that have been found,

- Crash (27 issues): This category includes bugs that cause an application or system to abruptly stop functioning. These issues are high priority due to their critical impact on users.
- Performance (14 issues): Performance bugs reduce application speed or responsiveness, often caused by inefficient algorithms, excessive resource usage, or concurrency problems.
- Inaccurate Status (11 issues): These bugs lead to inconsistencies between displayed information and actual system states, potentially resulting in user confusion.
- Semantic (9 issues): Semantic bugs are logical errors that cause incorrect behavior, even if the application does not crash.
- Hang (7 issues): This category includes bugs that make the system unresponsive due to deadlocks, infinite loops, or thread management issues.
- Resource Leak (6 issues): These bugs occur when applications fail to release resources properly, leading to performance degradation or crashes.
- Other Categories: Additional issues such as memory leaks, false positives, and overflow errors were also identified, though less frequently.

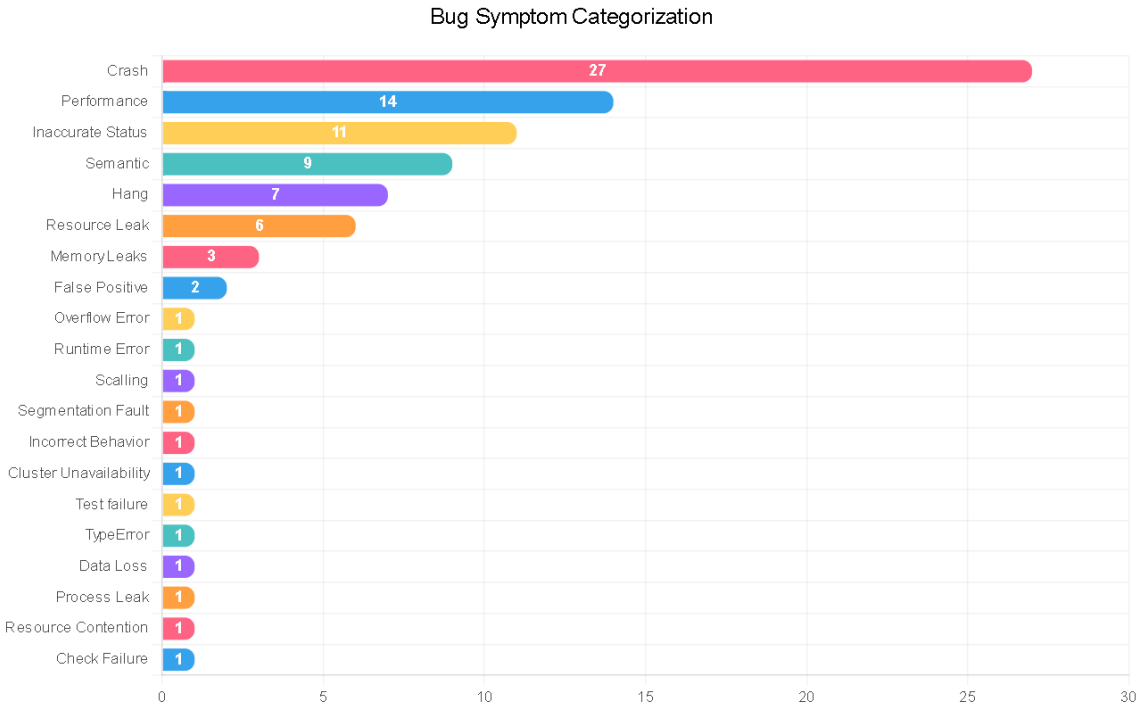

Figure 2. Bug symptom categorization

The results of the implementation of classification categorization based on bug symptoms in ray framework obtained several bug categories based on the number of occurrences that appear most often. The bug categories with more than ten occurrences are “Crash,” “Performance”, and “Inaccurate Status”. Based on the results of the study “A Comprehensive Study of WebAssembly Runtime Bugs” by Yue Wang, there is validation of the results of the implementation of bug symptom categorization in the ray framework, especially for the “Crash” category as one of the categories with the highest frequency. This study found that the most common symptom of bugs in WebAssembly runtime is “Crash”, which accounts for 56.86% of all WebAssembly runtime bugs [22]. This finding is in line with the results of the implementation of bug symptom categorization in ray framework, where the “Crash” category is one of the categories with the highest frequency of bug occurrence. This similarity can be explained by the fundamental nature of the runtime system, both WebAssembly and ray framework, which is responsible for executing and managing tasks or applications that run on it. Errors in the runtime system can cause failures or crashes in the executed applications or tasks, thus making “Crash” a common bug symptom.

Then another support is in the research “Performance Bug Analysis and Detection for Distributed Storage and Computing Systems” by Jiaxin and his team, emphasizing that bugs that cause blocking, namely

bugs that cause execution time to increase dramatically as workload size increases, are an important category of performance bugs in distributed storage and computing systems [23]. The "Performance" category identified in the implementation of bug symptom categorization can be supported by this study, given that blocking type bugs directly affect system performance. This suggests that identifying and addressing performance bugs, including blocking bugs, should be a priority in an effort to improve the reliability and performance of ray framework systems.

Plus the research "Vicious Cycles in Distributed Software Systems" by Shangshu and his team, where identified vicious cycles in distributed software systems, which are mainly caused by improper error handling, and suggested that monitoring tools as well as the use of exponential backoff can prevent the occurrence of such cycles [17]. These findings can support the identification of "Inaccurate Status" and "Performance" issues as major bug symptoms. The use of effective monitoring and exponential back off strategies can help in reducing the frequency of bugs related to inaccurate status and performance issues, thus strengthening the argument that the development of better error handling strategies and monitoring mechanisms can have a significant impact on the ray framework. The identification of dominant bug categories suggests that the ray framework requires targeted improvements in crash mitigation, performance optimization, and accurate state representation. Future research should explore advanced monitoring techniques and automated debugging tools to reduce the frequency of these issues.

3.2. Bug main characteristics results

This subsection explores key characteristics of the identified bugs, particularly focusing on the dependency between actor instances and bug occurrences within the ray framework.

Table 1. Bug main characteristics results

Characteristic	Instance of actor	Types of actors	Occurrence of failure	Ordering dependency	Configuration dependency
Count of yes	17	34	57	26	34
Count of no	74	57	34	65	57
Total	91				

3.2.1. Dependency of instance of actor in bug

The first column of Table 1 shows that 74 out of 91 identified bugs do not require multiple actor instances to occur, indicating that these bugs emerge independently within isolated actor operations. However, 17 issues exhibit a dependency on multiple actor instances, suggesting that concurrency and interactions between actors contribute to the emergence of certain bugs. The primary causes of these issues are logic errors in actor handling, such as improper reference management, data inconsistencies, and insufficient error handling.

Based on the results conducted on the ray framework, the majority of bugs do not depend on the instances of actors in the occurrence of bugs. Based on the results of the research "Actor concurrency bugs: a comprehensive study on symptoms, root causes, application programming interface (API) usages, and differences" by Bagherzadeh and his team, it can be validated that the majority of bugs in ray framework do not require the presence of multiple actor instances to appear. The study categorized actor concurrency bugs into five symptoms, ten root causes, and a small number of API packages. Where logic was the most common cause and untyped communication was the least common [24].

These findings support the results which show that ray framework bugs can occur in independent actor operations, without the need for interaction with other actor instances. This is because most bugs are caused by logic errors in the handling and management of the actors themselves, rather than from interactions between actors. Logical errors, such as improper reference handling, improper data management, or insufficient error handling, can cause bugs in individual actors, without involving other actors. For example, bugs such as improper argument handling during class instantiation, use of serialization protocols that do not support large data sizes, or inaccurate handling of actor states can all occur to a single actor without requiring interaction with other actors. Therefore, the implementation results showing that the majority of bugs in ray framework do not require a multiplicity of actor instances to occur are supported by Bagherzadeh and his team's research findings which state that logic errors are the most common cause of actor concurrency bugs, and these bugs can occur in individual actors without involving interactions between actors.

3.2.2. Dependency on types of actors in bugs

The second column of Table 1 reveals that 57 out of 91 bugs in the ray framework are independent of actor type diversity. These bugs occur in the execution of a single actor without requiring interaction with actors of different types. However, 34 issues exhibit dependencies on multiple actor types, indicating that interactions between distinct actor roles can trigger complex bug conditions.

The results highlighting the dependence on types of actors in the emergence of bugs in the ray framework, show that the majority of bugs do not depend on the diversity of actor types. This result is supported by the results of the study "A Comprehensive Study on Bugs in Actor Systems" by Hedden and his team, where it can be validated that the majority of bugs in ray framework do not depend on the diversity of actor types. The study analyzed a total of 126 actor-related bugs, and found that 57.5% of the bugs were common logic errors, while communication-related bugs were only 20.5% and coordination between actors was 22% [25]. These findings support the implementation results which show that bugs in ray framework can occur in a single execution of a particular actor, without requiring interaction with other actors. The majority of bugs are caused by general logic errors, such as null pointers, buffer optimization issues, or the system not terminating properly, which are not specifically related to communication or coordination between actors.

Although this study also identified bugs related to communication and coordination between actors, the percentage is smaller than the general logic errors. It can be concluded that this shows that most of the bugs in the ray framework can occur in individual actors, without involving interaction with other actors. For example, bugs such as improper reference handling, incorrect data management, or inadequate error handling can all occur on a single actor without requiring interaction with other actors. These bugs are mostly caused by logical errors in the handling and management of the actor itself, not from interactions with other actors.

3.2.3. Occurrence of failure in bug occurrence

The third column of Table 1 shows that 57 out of 91 bugs are reactive, meaning they arise as a direct response to system failures. These bugs are triggered by pre-existing errors or malfunctions. Conversely, 34 bugs occur independently of system failures, implying they stem from inherent implementation flaws rather than system breakdowns.

The results show that the occurrence of bugs in ray framework is influenced by system failure dependencies. This result is supported by the results of the research "Bug characteristics in open source software" by Lin Tan, where it can be validated that the majority of bugs in ray framework appear as a direct response to a system failure. This study found that semantic bugs are the dominant root cause in open source software, and most security bugs are caused by semantic bugs [26].

This finding supports the results which show that bugs in the ray framework are reactive, where they arise as a direct consequence of a failure or error that has occurred in the system. Semantic bugs, such as logic errors, improper data handling, or insufficient error handling, tend to appear in response to failure or error conditions that occur in the system. For example, bugs such as improper handling when actors are deleted, inadequate error handling by GCS when actors are deleted, or lack of synchronization between status and error messages, all of which arise in direct response to a failure condition or error occurring in the system.

These bugs are reactive in that they arise as a result of a pre-existing failure or error condition, such as an actor failure, node failure, or error in data management. The system then reacts to those failure conditions or errors in an incorrect way, causing semantic bugs to appear. Therefore, the implementation results showing that the majority of bugs in ray framework arise as a direct response to a system failure are supported by Lin Tan's research findings which state that semantic bugs are the dominant root cause in open source software, and these bugs tend to arise in response to failure conditions or errors in the system.

3.2.4. Ordering dependency in bug occurrence

The fourth column of Table 1 indicates that 65 out of 91 bugs are independent of event sequence. These bugs arise from general logic errors rather than specific concurrency or timing conditions. However, 26 bugs exhibit a dependency on event order, meaning that a precise sequence of interactions is necessary to trigger them.

The results show that bug occurrence in ray framework does not always require a specific sequence of events. Based on the results of the research "A Method and Tool for Finding Concurrency Bugs Involving Multiple Variables with Application to Modern Distributed Systems" by Zhuo Sun, there is a rejection of the implementation results which state that the majority of bugs in ray framework do not require a specific sequence of events to occur. The research states that distributed concurrency bugs often have simple causes and can be caught by simple tests, but are very difficult to trace and detect due to their complex non-deterministic nature [27].

This research focuses on atomicity violation, which is the most common type of distributed concurrency bug, and presents a model checking-based tool to predict distributed concurrency atomicity violation bugs in modern microservice-based distributed systems. The findings reject implementation results that state that the majority of bugs in the ray framework are generic and not related to specific concurrency or timing scenarios. Instead, this study emphasizes that distributed concurrency bugs, such as atomicity violations, are highly dependent on the sequence of events and the non-deterministic nature of the distributed

system. Therefore, the results of this study reject the implementation result that the majority of bugs in ray framework do not require a specific sequence of events and are general.

3.2.5. Configuration dependency in bug occurrence

The last column of Table 1 reveals that 62 out of 91 bugs are independent of specific configuration settings. These bugs arise due to implementation flaws rather than variations in system configuration. However, 29 bugs demonstrate a dependency on configuration settings, implying that specific system parameters can influence bug manifestations.

The results show that the emergence of bugs in ray framework does not always require special configuration. Based on the results of the research "Understanding and discovering software configuration dependencies in cloud and datacenter systems" by Qingrong Chen, there is a rejection of the implementation results which state that the majority of bugs in ray framework do not depend on certain configurations. This research presents a study of configuration dependencies in software and tools for discovering these dependencies [28].

This research defines five types of configuration dependencies and identifies common code patterns associated with these dependencies. The findings of this study show that configuration dependencies are very common and diverse in software, and prove that configuration dependencies can be discovered automatically. In addition, this study found 448 previously undocumented configuration dependencies, indicating that configuration dependencies are an important issue to consider in software engineering. Therefore, the results of this study reject the implementation results which state that the majority of bugs in ray framework do not depend on specific configurations.

Although the results state that the majority of bugs in ray framework do not depend on specific configurations, the findings of this study indicate that configuration dependency is a common and significant issue in software. Therefore, it is necessary to properly analyze and manage configuration dependencies to avoid potential bugs and ensure reliability and proper functionality in software systems.

3.3. Threats to validity

This subsection identifies potential threats to the validity of the study's findings, particularly in analyzing bugs within the ray framework using experimental analysis and bug abstraction methods. Addressing these threats is essential to accurately interpret and generalize the research results. The identified threats are as follows,

- The study focuses exclusively on the ray framework, which may limit the generalizability of the findings. The results could differ if applied to other distributed computing frameworks, such as faust or scalable concurrent operations in Python (SCOOP). The conclusions may not be universally applicable across all distributed systems.
- The formulation of questions for identifying bug characteristics is based on researchers' interpretations. Alternative methodologies for bug characterization may yield different results, indicating that subjective judgment could influence the research outcomes.
- The use of experimental analysis and bug abstraction as research methods may impact the study's validity. Different methodological approaches, such as case studies or statistical analyses, might produce varying insights, suggesting that the choice of methodology could affect the findings.
- The experiments were conducted in a predefined environment using Docker containers. While this setup ensures consistency, it may not fully reflect variations encountered in real-world deployments. Consequently, the results may not entirely represent performance in production environments.
- The dataset of bugs was compiled using specific selection criteria, such as closed issues and core bugs within the ray framework. This filtering process may introduce bias, potentially omitting certain types of bugs or issues that exist within the framework but were not included in the dataset.

By acknowledging these threats, the study provides a clearer perspective on its limitations, helping to contextualize the research findings and inform future studies seeking to enhance the reliability and applicability of bug analysis in distributed computing frameworks.

4. CONCLUSION

This study successfully classifies major bug categories and identifies key characteristics of bugs within the ray framework through experimental analysis and bug abstraction. Specifically, the classification of bug categories based on symptoms in the ray framework identified three primary categories from a dataset of 91 frequently occurring bugs: "Crash," "Performance," and "Inaccurate Status." This classification provides a structured understanding of common issues and serves as a foundation for developing targeted debugging strategies. Furthermore, the key characteristics of bugs were analyzed using bug abstraction. The

findings indicate that most bugs are not dependent on factors such as the number of actor instances, variations in actor types, event sequences, or specific configurations. Instead, these bugs predominantly arise as reactive responses to system failures, suggesting that their root causes are not directly linked to specific actor properties. This insight is crucial for improving debugging approaches and enhancing the robustness of the ray framework.

ACKNOWLEDGEMENTS

The author expresses sincere gratitude to Andrew Quinn for his invaluable guidance in shaping this research and ensuring its alignment with the study's objectives. The author also extends thanks to Institut Teknologi Del for providing funding and facilities that greatly supported the successful completion of this study.

FUNDING INFORMATION

The project was funded and supported by the Institute of Technology Del.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Arnaldo Marulitua Sinaga	✓	✓		✓		✓	✓	✓	✓	✓		✓	✓	✓
Wordyka Yehezkiel Nainggolan	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject administration
Fu : **Fu**nding acquisition

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

DATA AVAILABILITY

The data that support the findings of this study are available on request from the corresponding author, [Wordyka Yehezkiel Nainggolan]. The data, which contain information that could compromise the privacy of research participants, are not publicly available due to certain restrictions.

REFERENCES

[1] X. Wang, B. Shi, and Y. Fang, "Distributed systems for emerging computing: platform and application," *Future Internet*, vol. 15, no. 4, 2023, doi: 10.3390/fi15040151.
[2] R. Yosibash and R. Zamir, "Frame codes for distributed coded computation," *2021 11th International Symposium on Topics in Coding, ISTC 2021*, 2021, doi: 10.1109/ISTC49272.2021.9594259.
[3] T. H. Chang, M. Hong, H. T. Wai, X. Zhang, and S. Lu, "Distributed learning in the nonconvex world: From batch data to streaming and beyond," in *IEEE Signal Processing Magazine*, 2020, vol. 37, no. 3, pp. 26–38, doi: 10.1109/MSP.2020.2970170.
[4] Y. Sun, F. Zhang, J. Zhao, S. Zhou, Z. Niu, and D. Gunduz, "Coded computation across shared heterogeneous workers with communication delay," *IEEE Transactions on Signal Processing*, vol. 70, pp. 3371–3385, 2022, doi: 10.1109/TSP.2022.3185905.
[5] H. M. Tran, S. T. Le, S. Van Nguyen, and P. T. Ho, "An analysis of software bug reports using machine learning techniques," *SN Computer Science*, vol. 1, no. 1, 2020, doi: 10.1007/s42979-019-0004-1.
[6] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "TaxDC," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 517–530, 2016, doi: 10.1145/2980024.2872374.
[7] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, 2007, pp. 51–68.
[8] I. Neal *et al.*, "A gamotto: How persistent is your persistent memory application? This paper is included in the proceedings of the design and implementation a Gamotto: How persistent is your persistent memory application?," in *Osdi*, 2020, pp. 1047–1064.
[9] J. Lu, F. Li, C. Liu, L. Li, X. Feng, and J. Xue, "CloudRaid: Detecting distributed concurrency bugs via log mining and enhancement," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 662–677, 2022, doi: 10.1109/TSE.2020.2999364.

[10] E. Michael, D. Woos, T. Anderson, M. D. Ernst, and Z. Tatlock, "Teaching rigorous distributed systems with efficient model checking," 2019, doi: 10.1145/3302424.3303947.
[11] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, 2022, doi: 10.1145/3512345.
[12] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pp. 561–577, 2007.
[13] R. P. Team, "Ray: A general purpose framework for distributed computing," in *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, pp. 561–577.
[14] S. Chen, "The advance of distributed computing methods," *Highlights in Science, Engineering and Technology*, vol. 39, pp. 586–594, 2023, doi: 10.54097/hset.v39i.6594.
[15] H. Lin, X. Qin, S. Qiu, Y. Sun, Z. Yin, and W. Liu, "Ray-based elastic distributed data parallel framework with distributed data cache," *2023 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2023*, pp. 809–817, 2023, doi: 10.1109/IPDPSW59300.2023.00135.
[16] X. X. Ruan and C. C. Wu, "Boost the performance of model training with the ray framework for emerging AI applications," *Proceedings - 2022 IET International Conference on Engineering Technologies and Applications, IET-ICETA 2022*, 2022, doi: 10.1109/IET-ICETA56553.2022.9971626.
[17] S. Qian, W. Fan, L. Tan, and Y. Zhang, "Vicious cycles in distributed software systems," in *Proceedings - 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*, 2023, pp. 91–103, doi: 10.1109/ASE56229.2023.00032.
[18] S. A. Białowąs, A. Reshetkova, and A. Szyszka, "Experimental design and biometric research. Toward innovations," *Experimental design and biometric research. Toward innovations*, 2021, doi: 10.18559/978-83-8211-079-1.
[19] N. M. McKevett and R. S. Codding, "Brief experimental analysis of math interventions: A synthesis of evidence," *Assessment for Effective Intervention*, vol. 46, no. 3, pp. 217–227, 2021, doi: 10.1177/1534508419883937.
[20] J. A. Mellott and S. P. Ardoin, "Using brief experimental analysis to identify the right math intervention at the right time," *Journal of Behavioral Education*, vol. 28, no. 4, pp. 435–455, 2019, doi: 10.1007/s10864-019-09324-x.
[21] B. Millidge, "Towards a mathematical theory of abstraction," 2021, [Online]. Available: http://arxiv.org/abs/2106.01826.
[22] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *Proceedings - 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023*, 2023, pp. 355–366, doi: 10.1109/SANER56733.2023.00041.
[23] J. Li *et al.*, "Performance bug analysis and detection for distributed storage and computing systems," *ACM Transactions on Storage*, vol. 19, no. 3, 2023, doi: 10.1145/3580281.
[24] M. Bagherzadeh, N. Fireman, A. Shawesh, and R. Khatchadourian, "Actor concurrency bugs: A comprehensive study on symptoms, root causes, API usages, and differences," in *Proceedings of the ACM on Programming Languages*, 2020, vol. 4, no. OOPSLA, doi: 10.1145/3428282.
[25] B. Hedden and X. Zhao, "A comprehensive study on bugs in actor systems," 2018, doi: 10.1145/3225058.3225139.
[26] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014, doi: 10.1007/s10664-013-9258-8.
[27] Z. Sun, "A method and tool for finding concurrency bugs involving multiple variables with application to modern distributed systems," *FIU Electronic Theses and Dissertations*, pp. 1–99, 2018, [Online]. Available: https://digitalcommons.fiu.edu/etd/3896.
[28] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374, doi: 10.1145/3368089.3409727.

BIOGRAPHIES OF AUTHORS

Arnaldo Marulitua Sinaga earned his bachelor's degree in Informatics Engineering from Institut Teknologi Bandung, Indonesia. He holds a master's degree and a Ph.D. in Software Testing from the University of Wollongong, Australia. His research focuses on software testing, web development, and mobile application development, particularly in the areas of tourism, MSMEs, and government. Currently, he serves as the Head of the Del Software Quality Assurance and Testing Centre at Institut Teknologi Del, Indonesia. He can be contacted at aldo@del.ac.id.

Wordyka Yehezkiel Nainggolan earned his bachelor's degree in Software Engineering from Institut Teknologi Del, Toba, North Sumatra, Indonesia. He was involved in an international research collaboration with the University of California, Santa Cruz, USA, contributing to the project titled "Simple and Efficient Distributed Property Testing". His research work focused on developing high-level abstractions for specifying and evaluating distributed correctness properties within the Ray Framework. Wordyka's research interests include distributed systems, software testing, and backend development. He has also co-authored several technical reports and is committed to advancing knowledge in distributed computing. He can be contacted at if420041@students.del.ac.id.