Osaka University Knowledge Archive

Title	Study on Cost-Effective Debugging Methods under Restricted Resources
Author(s)	嶋利, 一真
Citation	大阪大学, 2022, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/88137
rights	
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

Study on Cost-Effective Debugging Methods under Restricted Resources

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2022

Kazumasa SHIMARI

List of Publications

Major Publications

1. Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Naoto Ishida, Katsuro Inoue, "NOD4J: Near-omniscient debugging tool for Java using size-limited execution trace," Science of Computer Programming, Vol.206, 102630 (13 pages), June 2021.

2. Kazumasa Shimari, Takashi Ishio, Katsuro Inoue, "A clustering-based filtering method for similar source code fragment search," IEICE TRANSACTIONS on Information and Systems, Vol.J103-D, No.10, pp.751-753, October 2020 (in Japanese).

3. Kazumasa Shimari, Takashi Ishio, Katsuro Inoue, "An execution trace recording method using a limited size storage for Java," JSSST Computer Software, Vol.36, No.4, pp.107-113, October 2019 (in Japanese).

4. Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, Katsuro Inoue: "Near-Omniscient Debugging for Java using size-limited execution trace," In Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, pp.398-401, Cleveland, OH, USA, October 2019.

Related Publications

1. Sakutaro Sugiyama, Takashi Kobayashi, Kazumasa Shimari, Takashi Ishio: "JISDLab: A web-based interactive literate debugging environment," 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (to appear).

2. Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, Katsuro Inoue: "PADLA: A dynamic log level adapter using online phase detection," In Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension, pp.135-138, Montreal, Quebec, Canada, May 2019.

3. Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, Katsuro Inoue : "A method for dynamically adjusting the amount of logging using phase detection for program execution," In Proceedings of the IPSJ/SIGSE Winter Workshop, pp.17-18, Iizaka, Fukushima, Japan, January 2019 (in Japanese).

4. Kazumasa Shimari, Takashi Ishio, Katsuro Inoue : "Generating test cases from execution traces to support dependency updates," In Proceedings of

the IPSJ/SIGSE Winter Workshop, pp.22-23, Miyajima, Hiroshima, Japan, January 2018 (in Japanese).

5. Kazumasa Shimari, Takashi Ishio, Katsuro Inoue : "Prototype of a low invasive monitoring tool for analyzing software execution," In Proceedings of the IPSJ/SIGSE Software Engineering Symposium, pp.224-227, Waseda, Tokyo, Japan, September 2017 (in Japanese).

Abstract

In recent years, software development costs have increased, and the debugging costs, which account for half of software development costs, have also increased accordingly. Many debugging methods have been proposed to reduce debugging costs, but debugging methods that cannot predict the required resources in advance may lead to unexpected increases in debugging costs, which in turn may lead to higher development costs. Therefore, developers requires cost-effective methods which can debug the problem under restricted resources. In this dissertation, we propose debugging methods that improve the current software debugging methods with respect to resources, enabling debugging with resources that are acceptable to the developers.

In the first part of this dissertation, we propose Near-Omniscient Debugging which monitors a Java program's execution within limited storage space. We confirm the usefulness of Near-Omniscient Debugging by measuring the data dependencies' accuracy of the execution traces, the amount of execution traces, and the percentage of bugs whose bug-related instructions are completely recorded. Using this debugging method, we have implemented a tool named NOD4J which annotates the source code with observed values in an HTML format. We show two examples that our tool can debug defects using incomplete execution traces.

In the second part of this dissertation, we tackle the test case selection problem. Regression testing is often performed to verify the software compatibility before and after a change. However, a large number of test cases in regression testing steps take a lot of time to verify the compatibility and diagnose the failure. To solve this problem, we have developed a method to select the necessary tests for behavior verification based on the similarity of runtime information. We have evaluated the coverage and diversity of execution time of the proposed method.

In the third part of this dissertation, we propose a filtering method for the result of the similar source code fragments search. When developers search similar buggy source code fragments by a similar source code fragment search tool, developers should inspect all source code fragments in search results if they have enough resources, but development resources are restricted in general. To deal with this problem, our filtering method groups code fragments by applying clustering to search results and excludes code fragments with a low probability of correct answers from the output. We have confirmed the usefulness of our filtering method on a dataset of similar code fragments containing bugs.

Using these proposed debugging methods, developers can reduce debugging costs and perform debugging tasks with restricted resources.

Acknowledgement

I would like to express my most sincere gratitude to Professor Katsuro Inoue for his continuous support and guidance. Without his tremendous help, experience and advice, this thesis would not have been completed.

I would like to express my sincere gratitude to Professor Fumihiko Ino and Professor Shinji Kusumoto for their guidance and advice in writing this paper. I would also like to acknowledge the guidance of Professors Toshimitsu Masuzawa, Yasushi Yagi and Hajime Nagahara.

I would like to express my deepest gratitude to Associate Professor Takashi Ishio at Nara Institute of Science and Technology for his guidance, advice and support in this research.

I would like to express my gratitude to Associate Professor Makoto Matsushita and Specially Appointed Professor Shusuke Haruna for their guidance and support in this research.

I am very grateful to Assistant Professor Tetsuya Kanda for his guidance and assistance in daily research activities and discussions.

I wish to thank Associate Professor Norihiro Yoshida at Nagoya University and Assistant Professor Raula Gaikovina Kula at Nara Institute of Science and Technology for their advice during my research activities.

I would like to thank Kaoru Ito at Japan Research Institute, Yuji Fujiwara and Davide Pizzolotto at Osaka University for their active research discussions.

I would like to express my gratitude to all the members of Inoue Laboratory for their advice and cooperation in my daily research life.

Finally, I wish to thank my family for supporting my school life.

Contents

1 Introduction **1**
1.1 Software Debugging Cost 1
1.2 Debugging Technique 2
1.3 Research Overview 3
1.4 Outline 5

2 An Execution Trace Recording Method Using a Limited Size Storage for Java **7**
2.1 Introduction 7
2.2 Related Work 8
2.2.1 Execution Trace Reduction 8
2.2.2 Dynamic Analysis with Low Overhead 8
2.3 Proposed Method 9
2.3.1 The Model of Execution Trace 9
2.3.2 Recording Method of Execution Trace 9
2.4 Evaluation on Trace Quality 11
2.4.1 Percentage of Instructions where Values are Completely Recorded 12
2.4.2 Data Dependency Accuracy 12
2.5 Evaluation on Effectiveness for Debugging 15
2.5.1 Experimental Settings 15
2.5.2 RQ1. How is the runtime overhead of Near-Omniscient Debugging? 18
2.5.3 RQ2. Are the execution traces recorded by Near-Omniscient Debugging useful for debugging actual bugs? 22
2.6 Discussion 24
2.7 Conclusion and Future Work 25

3 NOD4J: Near-Omniscient Debugging Tool for Java Using Size-Limited Execution Trace **27**
3.1 Introduction 27
3.2 Background 28
3.3 NOD4J Overview 29
3.3.1 Trace Recorder 29
3.3.2 Post Processor 31
3.3.3 Interactive View 33
3.3.4 Usage 34
3.4 Usage Examples 36

3.4.1 Use Case 1: Lang 2b 37
3.4.2 Use Case 2: Math 59b 41
3.5 Discussion 42
3.6 Conclusion 43

4 Effective Test Case Selection Based on Similarity of Runtime Information for Dependency Update 45
4.1 Introduction 45
4.2 Background 46
4.2.1 Dependency Compatibility 46
4.2.2 Test Case Selection 47
4.3 Our Test Selection Method 48
4.3.1 STEP I: Collect Runtime Information 48
4.3.2 STEP II: Classify Similar Execution 49
4.4 Case Study 50
4.4.1 RQ1: Does our test selection method provide better coverage than the existing method? 51
4.4.2 RQ2: How diverse is the execution time? 51
4.4.3 RQ3: Can our method classify an existing bug-related test case? 51
4.5 Field Experiment 52
4.6 Conclusion 53

5 A Clustering-based Filtering Method for Similar Source Code Fragment Search 55
5.1 Introduction 55
5.2 Proposed Method 56
5.3 Evaluation 56
5.4 Conclusion 58

6 Conclusion and Future Work 61
6.1 Summary of Studies 61
6.2 Future Work 62

List of Figures

2.1 An Execution Trace of Near-Omniscient Debugging which Records k Events for Each Instruction. 10
2.2 The Size of Execution Trace Recorded by Near-Omniscient Debugging. 12
2.3 Percentage of Instructions Where Values are Completely Recorded. 13
2.4 Distribution of the Percentage of Execution Trace Size Occupied by Top N% Instructions for Each Bug ID When Instructions are Sorted in Descending Order of Execution Counts. 17
2.5 The Distribution of Execution Time for Each Bug ID. 18
2.6 The Distribution of the Size of Execution Trace for Each Bug ID at Each Buffer Size. 19
2.7 The Distribution of the Incomplete Recording Rate of Bug-Related Instructions for Each bug IDs at Each Buffer Size When Complete Recording Fails. 24

3.1 An Example of Logging Code Locations for Recording Method Execution and Call Events. 30
3.2 Trace View. 33
3.3 Trace View with Filtering. 33
3.4 The Welcome Page of NOD4J. 36
3.5 The Browser Page of NOD4J. 36
3.6 The Interactive View for the Current Scenario. 37
3.7 The Result of the Filtering. 37
3.8 A Test Method for `LocaleUtils.toLocale` Method in Lang 2b. . 38
3.9 Actual Parameter Values that Induced a Failure in Lang 2b. 38
3.10 A Filtering Result of the `toLocale` Method in Lang 2b (The First Half of the Method). 39
3.11 A Filtering Result of the `toLocale` Method in Lang 2b (The Second Half of the Method). 40
3.12 Failure Induced Method at Math 59b. 42

4.1 An Overview of Our Method. 49
4.2 The Distribution of the Execution Time without Outlier. 52

5.1 Harmonic Mean of the Proposed Method and Baseline. 58

List of Tables

2.1 Size of Benchmark Programs and Their Execution Trace. 11
2.2 Accuracy of Data Dependencies Obtained by Near-Omniscient Debugging (Sum of All Benchmarks). 13
2.3 Accuracy of Data Dependencies Obtained by Time Series Recording (Sum of All Benchmarks). 14
2.4 Accuracy of Data Dependencies Obtained by Near-Omniscient Debugging (Each Benchmark, $k = 16$). 14
2.5 Accuracy of Data Dependencies Obtained by Time Series Recording (Each Benchmark, $k = 16$). 15
2.6 The Defects4J Dataset Used for Evaluation. 16
2.7 The Execution Cost of Bugs with the Largest Execution Trace for Each Project. 21
2.8 Complete Recording Rate of **Bug-Related Instructions** at Each Buffer Size. 22
2.9 The Number of Bug IDs Recording **Bug-Related Instructions** Completely and with Trace Reduction at Each Buffer Size. 22
2.10 The Number of Bugs Including Incompletely Recorded **Bug-Related Instructions** at Each Buffer Size. 23

3.1 Major Runtime Events Recorded by the Recorder Component. . . 29
3.2 Events Linked to Identifiers in Source Code. 32
3.3 The Pseudo Variables Representing Invisible Values on Source Code. 32

4.1 Subsystem of the Target Simulator Under Test. 50
4.2 Result of the Coverage. 51

5.1 Dataset for Similar Source Code Fragments. 57
5.2 Filtering Accuracy of the Proposed Method and Baseline. 57

Chapter 1

Introduction

1.1 Software Debugging Cost

In recent years, a lot of software has been developed, and the resources required for software development have become enormous. According to a survey on software development [1], the global cost of software development is 1.25 trillion dollars, and according to Evans Data Corporation [2], there are 26.9 million software developers in the world. A huge amount of resources are required for software development tasks, but required resources vary greatly depending on the scale of the software development. Developers usually estimate the cost before software development and develop the software with restricted human resources and time in general. In order to develop software under these constraints, cost-effective development methods are indispensable.

In the software development process, debugging is particularly essential, time-consuming and expensive. Software development is divided into several processes by the waterfall model consisting of (1) requirements definition, (2) system and software design, (3) implementation and unit testing, (4) integration and system testing, and (5) operation and maintenance [3]. The debugging process happens after the developer has written the code, and in the example of the waterfall model described above, it is performed in (3) through (5). If a bug occurs in a later process, developers have to fix the code and repeat the same process for verification, so the cost will surge [4]. Therefore, effective debugging is essential to find and fix the problems quickly.

A recent study showed a cost distribution software developments [1]. This study revealed that the tasks related to debugging, such as bug fixing and making code work, account for 50% of the cost, which was the same as the cost related to product development, such as designing work and writing code. This study also showed that the total estimated global cost of debugging, including wages and overheads, was 312 billion dollars per year. As the scale of software has grown in recent years, the complexity has increased, and finding and fixing bugs has become difficult [5].

Hardware costs are also enormous in debugging using detailed information. Long-running systems such as Web services generate huge amounts of logs, which may consume huge amounts of disk space. For example, Microsoft's service system can generate tens of terabytes of logs in a day [6]. Large-scale software systems

which run on a 24-hour basis also generate 50 GB of logs per hour [7]. In addition, in recent years, machine learning has become more and more popular, and required resources such as memory are larger.

Not putting enough resources into debugging due to its high cost will lead to greater losses in the future. There have been many cases where software bugs have caused programs to fail to run, resulting in huge losses [8] [9] [10]. National Institute of Standards and Technology (NIST) reported that the annual impact of software bugs and errors on the U.S. economy is estimated at 59.5 billion dollars (about 0.6% of GDP) [11]. For these reasons, there is a need in software development for a debugging method that can efficiently fix bugs while reducing the cost of debugging.

1.2 Debugging Technique

The debugging process is to analyze a program that does not meet the specification and in some cases to extend it (with debugging statements) to find a new program that is closer to the original program and meets the specification [12]. In other words, the purpose of debugging is to find out the problematic code that is causing the failures and fix them. In general, the debugging process can be decomposed into seven steps [13].

1. Track the problem in the database.
2. Reproduce the failure.
3. Automate and simplify the test case.
4. Find possible infection origins.
5. Focus on the most likely origins.
6. Isolate the infection chain.
7. Correct the defect.

Developers have to perform a lot of steps to correct the defect. A huge amount of manual work in debugging has a significant impact on the increase of debugging cost. The most costly steps are the identification of defects from Step 4 to Step 6 [13] [14]. Especially for large-scale software, failure analysis becomes more time-consuming due to the dependencies' complexity. Many debugging support methods and tools have been proposed to reduce the cost of manual work. We list up typical debugging techniques based on the existing survey [15] and describe these techniques.

Logging

Logging is used to record runtime information such as error messages and actual values of variables. Existing researches tackled the appropriate logging, which means where-to-log, what-to-log, and how-to-log [16]. Appropriate logging enables developers to get the information required to diagnose the cause of failure.

Detailed logging incurs system runtime overhead such as CPU consumption and I/O operations, but without detailed logging, necessary runtime information may be missing [17]. To solve this problem, logging libraries such as Apache Log4j 2 [18] control the number of log messages recorded to disk by setting the verbosity level for each logging statement according to their importance.

Assertions
Developers can specify assertions in the program code as conditional statements that terminate execution if they are evaluated as false, thereby automatically detecting software defects at runtime.

Breakpoint Debugging
Developer set breakpoints in the program to stop the program execution temporarily and check the variable values at that time [19]. Any variable can be analyzed by using single-step execution or by setting multiple breakpoints. In order to set breakpoints for analyzing the causes of failures, it is necessary to understand the behavior of the program such as execution order. Beller et al. reported that the IDE-provided debugging infrastructure was used about 80% when debugging [20]. This method is often used after understanding the outline of the program behavior by logging as described above.

Profiling
Profiling is to analyze metrics such as execution speed and memory usage at runtime. This method is used for debugging such as identifying abnormal behavior such as memory leaks [21].

Omniscient Debugging
Omniscient Debugging is a technique to record all the runtime events during program execution [22]. This technique enables developers to inspect the state of a program at an arbitrary point by simulating step-by-step execution in both the forward and backward. This method requires a huge amount of storage costs to record execution traces.

Test Case Prioritization and Selection
Software developers may prioritize and select their test cases to reduce the cost of regression testing [23]. However, manual test selection may miss bugs or waste time [24], so it may lead to an increase in software debugging costs. Previous studies have shown that automated test selection and prioritization techniques can significantly improve the failure detection rate [25] [26] [27]. If test cases that are likely to cause software failures can be appropriately prioritized and selected among many test cases, developers can eliminate the causes of failures in a short time without repeatedly executing similar test cases.

1.3 Research Overview

Existing debugging methods have successfully reduced debugging costs. However, since it is difficult to predict how much resources will be required for debugging tasks, debugging methods that require sufficient resources may not be applicable for debugging task.

In this dissertation, we discuss cost-effective debugging methods under restricted resources. We propose some debugging methods to make current software debugging methods available to those with restricted resources.

The first problem is the hardware costs of the Omniscient Debugging. Omniscient Debugging has to record a huge amount of execution traces to reproduce the execution. For example, one method records about 10MB of execution traces for each second of a Java program execution [28]. It is difficult for developers to estimate the size of the execution trace recorded before execution. Infinite loops during debugging can lead to a lack of disk space, which can cause bugs and consume more debugging resources.

To record the detailed program behavior within limited storage space constraints, we propose Near-Omniscient Debugging, a methodology that records an execution trace using fixed-size buffers for each observed instruction. We apply our method to benchmark applications and confirm that our method records values completely for most instructions while significantly reducing the amount of execution trace. In addition, our method records the data dependencies with high accuracy. We have applied the proposed method to actual bug datasets and confirmed that it was able to completely record the bug-related instructions in most of the bugs. We have also implemented the tool for Near-Omniscient Debugging. Our tool monitors a Java program's execution within limited storage space constraints and annotates the source code with observed values in an HTML format. Developers can easily investigate the execution and share the report on a web server. We show two examples that our tool can debug defects using incomplete execution traces.

The second problem is a test case selection for updating dependencies. In enterprise software development, user execution logs may be recorded in a rerunnable form, and these logs may be used for regression testing. Regression testing is performed to verify the software compatibility before and after a change. A large number of test cases in regression testing steps take a lot of time to verify the compatibility and diagnose the failure. Test case selection methods are proposed to reduce the number of the executed test cases using similarity of the test results [29] [30]. However, in order to use the historical data of test execution for selecting test cases, it is necessary to use the information from the repeated execution of a large number of test cases. Test case selection methods focusing on a user's source code change [26] [27] also cannot be applied to update dependencies because this task does not change the user's source code. Thus, it is difficult for existing research to select a small number of test cases from a large number of ones for updating dependencies.

We have developed a method to select the necessary tests to verify the compatibility of dependencies based on the similarity of runtime information. This method collects the executed instructions and their execution count in each test. Vectorizing this information and calculating the similarity between these executions, developers can select test cases with different types and instruction execution counts. As a result of the evaluation, we confirmed that the test cases selected by our method achieved higher coverage and more diverse execution times than baselines. The developers recognized the selected test cases as effective because they tested a wider range of parameters than their manually selected test cases. We selected useful tests for compatibility testing with the same number of tests as before and succeeded in reducing the cost of regression testing.

The third problem is inspecting whether source code fragments are buggy with restricted debugging resources. To search similar buggy source code fragments, a tool named NCDSearch is proposed [31]. This tool misses fewer source code fragments that include similar bugs compared to existing code search tools, but this tool also finds many source code fragments that do not include bugs. Developers should inspect all source code fragments in search results if they have enough resources, but development resources are restricted in general. This problem is similar to test case selection and prioritization, where developers need to identify code that is likely to include bugs in a restricted amount of time and resources. Therefore, it is necessary to reduce the debugging cost by prioritizing the presented source code fragments.

Here, we propose a filtering method for the result of the similar source code fragments search. This method applies clustering to the output results of similar source code fragment search tools to group code fragments. It excludes code fragments with a low probability of correct answers from the output. We confirmed the usefulness of the proposed method on a dataset of similar code fragments including bugs. This filtering allows developers to focus their debugging efforts only on the code that is most likely to include bugs, which contributes to reducing the debugging costs.

1.4 Outline

The rest of the dissertation is structured as follows:

In Chapter 2, we describe the details of Near-Omniscient Debugging method using a limited size storage. In Chapter 3, we describe a visualization method for execution traces obtained by the recording method proposed in Chapter 2, and the implementation of the recording and visualization methods. In Chapter 4, we describe a test selection method that uses runtime information to update dependencies. In Chapter 5, we describe a filtering method for similar source code fragment search. Finally, in Chapter 6, we summarize this dissertation and describe the future work.

Chapter 2

An Execution Trace Recording Method Using a Limited Size Storage for Java

2.1 Introduction

Debugging is the activity of identifying defects in the source code that cause software failures from externally observable symptoms. Observing the execution order of instructions and variable values is important for effective debugging [32]. In particular, it is useful to observe the program execution with and without failures and to investigate the differences in conditional branches and variable values at various points during the execution [33] [34].

Logging is widely used by developers as a means of observing the execution of software. This method can output messages that indicate the progress of the software process and important data outside the program [35]. However, logging recorded in production may not contain sufficient data for debugging since the data is specified at development time [36]. It is possible to design logging to record many variable values in advance, but as the amount of data to be recorded increases, it is necessary to pay attention to operational aspects such as the management of storage [37]. To enable efficient debugging, an automatic method capable of recording a program's execution in detail is needed.

Omniscient Debugging[22] is proposed as one of the methods to automatically and comprehensively collect the variable values needed for debugging, which records the complete memory state in a time series during the program execution. Using this method, it is possible to reproduce the internal program state at arbitrary points in time on a computer and observe the execution order of instructions and the variable values. However, this approach requires recording about 10MB of execution traces for every second of Java program execution [28]. It is difficult for developers to estimate the execution trace size before execution. Therefore, this implies difficulty in determining what data should be logged to fix bugs in a

deployed environment [38].

In this chapter, we propose Near-Omniscient Debugging to record an execution trace within limited storage space constraints. Since a full execution trace includes many uninteresting method calls such as utility functions [38], we introduce a parameter k that specifies the maximum number of recorded values for each instruction. This parameter limits the size of an execution trace for repeatedly executed instructions while keeping all actual values of variables associated with instructions that are executed less than k times. In concrete, by preparing a separate buffer of size k for each instruction, all observed values for instructions that have been executed less or equal than k times are kept, and for instructions that have been executed more than k times, only the most recent k times are kept. By counting up the number of instructions that constitute a program before execution, it is possible to set the value of k according to the storage area owned by the developer and to record the execution trace.

We will describe related work in Section 2.2 and the proposed method in Section 2.3. We evaluate the ability of the execution trace to record information in Section 2.4 and also evaluate on effectiveness for debugging actual bugs in Section 2.5. We discuss the limitation of Near-Omnsicient Debugging in Section 2.6. Finally, Section 2.7 summarizes and discusses future work.

2.2 Related Work

2.2.1 Execution Trace Reduction

Repetition of program instructions leads to larger execution traces. To reduce the size of execution traces, compression and sampling methods have been proposed.

Wang et al. [39] proposed an effective compression method tailored for execution traces comprising a sequence of memory addresses accessed by a program. This method employs delta encoding because programs often repeat the same instructions manipulating consecutive data locations in memory. Although the compressed trace is applicable to dynamic data-flow analysis, this method is unsuitable for recording the concrete runtime values of variables. In addition, the trace's size is hard to estimate prior to execution.

Cornelissen et al. [38] reported that execution traces excluding unimportant utility functions retain more information than a trace filtered by a simple sampling algorithm using the same storage space. The method does not directly reduce the size of an execution trace because it assumes that a full execution trace is recorded and filtered for each analysis. Our method reduces the repetition of data during the recording process.

Hizrel et al. [40] proposed Bursty Tracing, a sampling method that periodically turns monitoring on and off. This method can record rich information about program control flow compared with other sampling techniques. However, it is not designed to collect values of variables. This method also cannot estimate the size of a trace prior to execution.

2.2.2 Dynamic Analysis with Low Overhead

Another approach to minimizing runtime overhead and storage space is a specialized execution trace tailored for a specific purpose. Liu et al. [41] proposed an

analysis method to collect detailed information about suspicious behavior such as buffer overflows and memory leaks using lightweight memory access monitoring techniques. Zhang et al. [42] proposed a method to analyze data dependencies by converting execution traces to the program slice dynamically, which reduces storage usage drastically. Since those approaches are specialized for their purposes, they are inapplicable to variable values.

2.3 Proposed Method

We propose Near-Omniscient Debugging which records a partial execution trace of a program, which contains the latest k times trace at each instruction of Java bytecode. By keeping the most recent execution, the values of the abnormal behavior are likely to be recorded in an abnormal event such as program crashes. On the other hand, since instructions that are executed only once, such as at the start of program execution, are not discarded, it is possible to confirm the environment settings at program execution.

2.3.1 The Model of Execution Trace

The execution trace of Near-Omniscient Debugging is based on the existing trace recorder for REMViewer [43], an Omniscient Debugging tool. An execution trace includes (1) method entry and exit events with their arguments, return values, and exceptions and (2) values read from and written to local variables, fields and arrays. The recorder component assigns an object ID to distinguish each object reference. For objects, we store the ID value to distinguish the reference. For method invocations without arguments or return values, one data is recorded for each occurrence of the start and end of the invocation. These values may include redundant information, such as pairs of values written to and read from a field. However, considering the possibility of simultaneous operations by multiple threads and the possibility of access from native code or libraries that are outside the scope of observation, each value is recorded separately.

Our method represents such an execution trace as a series of observed values. An execution trace is a sequence of events $\langle d, t, v \rangle$ where d represents a data element (e.g., a local variable) used by an instruction, t represents a thread of control that executed the instruction, and v represents an observed value. An instruction identifies the Java bytecode instruction that caused the data exchange and the type of data. For example, if two different values, such as a method argument and a return value, are observed for a single instruction, they are considered different instructions.

2.3.2 Recording Method of Execution Trace

Near-Omniscient Debugging records the execution trace so that only the latest k values are kept when the instructions d are the same. During the program execution, the method prepares a buffer of length k for each instruction d, and keeps the observed values and the order of instructions in the entire execution in the memory. At the end of the program, the accumulated data is saved together by using the shutdown hook mechanism of the Java Virtual Machine. Compared

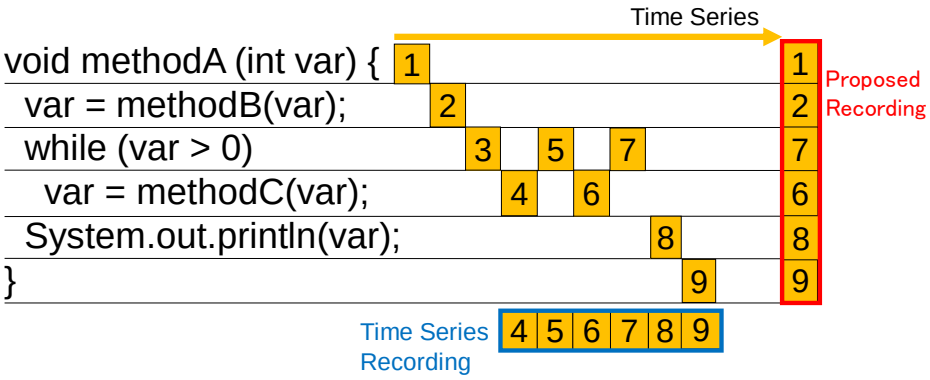

Figure 2.1: An Execution Trace of Near-Omniscient Debugging which Records k Events for Each Instruction.

to the execution traces of previous studies such as Matsumura et al.'s [43], it is equivalent to storing all observed values in a time series.

Figure 2.1 illustrates the key idea of Near-Omniscient Debugging. In this figure, we have a limited storage space that can record up to six steps of program execution. Suppose an execution comprises nine steps, as indicated by numbers in yellow boxes. A naive time series logging records the last six steps as indicated at the bottom of the figure. On the other hand, our method prepares buffers for each line of code to record the latest step for each line, as shown on the right side of the figure. This approach discards an execution trace for repeated instructions in the loop but retains the other information completely. By recording the latest observed values, abnormal behaviors are likely recorded in case a program crashes. Since the trace retains the program's initialization process which is executed only once, developers also can analyze the configuration parameters of the execution.

Since the number of instructions in the program m can be statically counted from the Java bytecode of the program, the storage cost required to store the execution trace $N = k \times m$ can be estimated in advance. A technique to generate bytecode dynamically is used in Java, but it does not occupy a large proportion of the program in normal use. We consider that it is possible to set the estimated value of m with a margin.

In this implementation, bytecode instrumentation is performed when the Java program loads the class, and the instruction for observation is embedded while assigning the instruction ID to the target program. The implementation uses 4 bytes for storing the instruction ID and thread ID and 8 bytes for storing the value, thus requiring 16 bytes of storage space for each value. The amount of additional information, such as program instruction information, depends only on the number of classes loaded as a program and the number of bytecode instructions. To preserve the execution order, this method records 8 bytes of indices simultaneously at runtime, so this method uses 24 bytes of memory for each recorded value and an array to manage them.

For string data, it is conceivable to store useful information about objects recorded as observed values by recording a fixed length up to a certain length and by recording a hash value of the contents. However, we decided that the appropriate way to store strings was a future issue because of the wide variety

Table 2.1: Size of Benchmark Programs and Their Execution Trace.

Benchmark	#Class	#Method	#Dataid	#Executed Dataid	#Observed Values	#Data Size
avrora	527	3,456	87,736	38,772	7,880,412,751	117.4 GB
batik	1,122	9,163	218,775	72,630	601,350,099	9.0 GB
fop	1,198	10,401	318,795	127,226	325,806,544	4.9 GB
jython	2,244	23,342	670,054	214,300	5,461,124,807	81.4 GB
luindex	231	2,467	73,094	25,838	1,542,871,620	23.0 GB
lusearch	199	2,140	58,127	16,256	5,503,253,333	82.0 GB

of strings used in the program. In the current implementation, the contents of strings are stored completely.

2.4 Evaluation on Trace Quality

We evaluate how much information the execution trace recorded by Near-Omniscient Debugging can store in limited storage size from the following two perspectives.

1. Percentage of instructions where values are completely recorded
2. Percentage of preserved data dependencies

The former is the probability of recording all the data for randomly selected instructions. We evaluate our method's usefulness in situations where the developer does not know in advance which instructions will be required. In the latter case, we evaluate whether the order relationship of the data flow is correctly preserved when the execution trace is considered as a series.

To evaluate the execution traces, we used 6 of the 14 benchmarks from DaCapo Benchmarks version 9.12-bach [44], which were confirmed to work properly. The execution traces include the calls to the standard library in the benchmarks but do not record the operations in the Java standard library. Table 2.1 shows the program size and the execution trace size for each benchmark. Since local variables values can be reproduced by computation from data received from outside the method, operations on local variables other than arguments are not measured in this evaluation. $\#Class$ is the number of classes loaded to run the benchmark, excluding the Java standard library (identified by package names such as `java`, `javax`, `sun`, etc.). $\#Method$ and $\#Dataid$ are the values counted for those classes, including those corresponding to methods and instructions that were not executed. $\#ObservedValues$ is the value when the complete execution trace is recorded, and $DataSize$ is the storage size when the complete execution trace is saved as 16 bytes per observed value.

We set the parameter k to $16, 32, 64, 128, 256$ for Near-Omniscient Debugging. Figure 2.2 shows the data size recorded by Near-Omniscient Debugging for each value of k. ALL in the figure is the data size when the complete execution trace is recorded. Near-Omniscient Debugging retains k values of 16-byte per instructions. Therefore, when jython is run with $k = 256$, the maximum number of instructions $670,054 \times\ 256 \times 16 = 2.7 \times 10^9$, which means that a maximum of about 2.6 GB of execution trace is recorded. However, since not all instructions are executed, the actual value is smaller. The data size recorded by Near-Omniscient Debugging

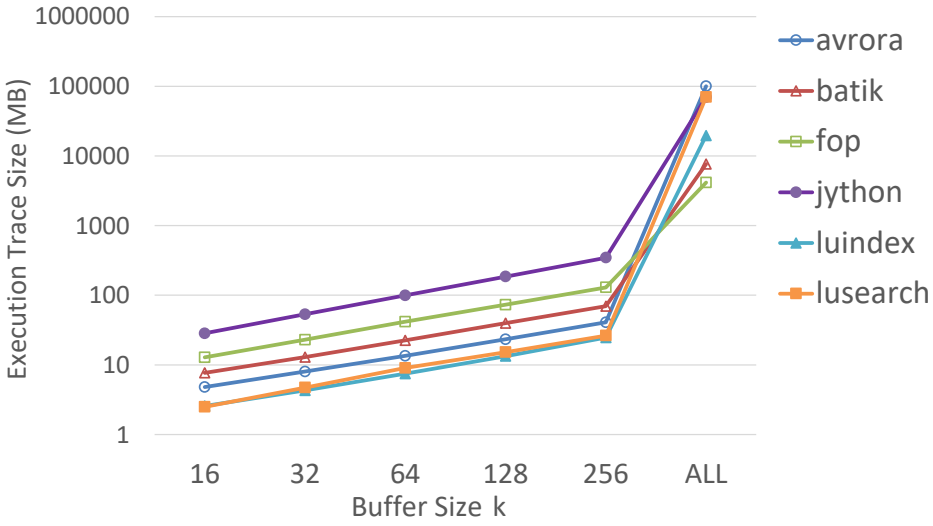

Figure 2.2: The Size of Execution Trace Recorded by Near-Omniscient Debugging.

with $k = 256$ is less than 1.0% on average compared to the case where the complete execution trace is recorded.

As a baseline for performance in an environment where the available storage space is limited, we use a time series-based method: a method that stores the latest N values in the entire execution trace. We refer to Near-Omniscient Debugging as Near-Omniscient Debugging and the baseline method as Time Series Recording.

2.4.1 Percentage of Instructions where Values are Completely Recorded

Figure 2.3 shows the difference in the percentage of instructions that can record values completely (i.e., the number of instruction execution count is less than or equal to k) out of the executed instructions for each benchmark when the number of values to be stored for each instruction k is varied.

Near-Omniscient Debugging records complete data for 60% of the instructions when $k = 16$, for 74% of the instructions when $k = 256$, and for the remaining instructions, it records values for the latest k times. Thus, even for program executions that would result in huge execution traces with conventional methods, complete information on many statements can be referenced with a limited amount of data (less than 1%). In contrast, Time Series Recording records data only for about 10% of the instructions, and the user cannot refer to the complete values for most of the instructions.

2.4.2 Data Dependency Accuracy

In order to evaluate whether Near-Omniscient Debugging correctly records the order relation regarding the data flow, we compare the data dependencies calculated from the execution traces of Near-Omniscient Debugging with those calculated from the full execution traces.

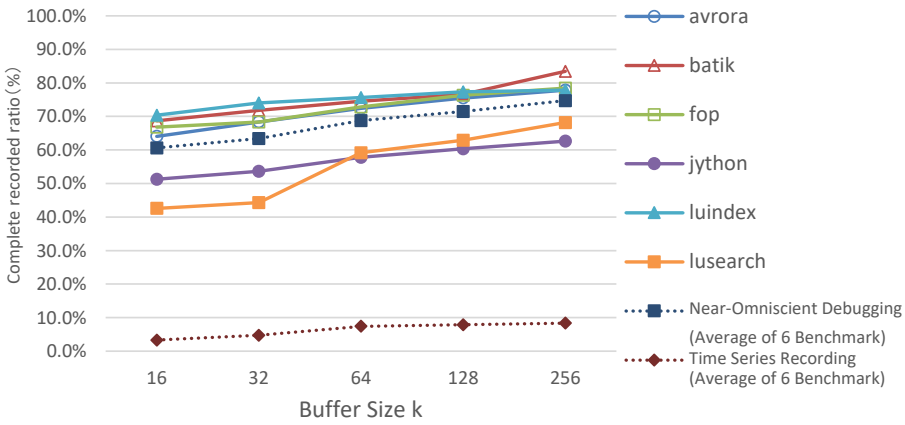

Figure 2.3: Percentage of Instructions Where Values are Completely Recorded.

Table 2.2: Accuracy of Data Dependencies Obtained by Near-Omniscient Debugging (Sum of All Benchmarks).

	Correct	Near-Omniscient Debugging			
k	Dependency	Dependency	Precision	Recall	F-measure
16	52,061	42,603	0.914	0.748	0.823
32	52,061	45,115	0.903	0.782	0.838
64	52,061	47,472	0.892	0.813	0.851
128	52,061	49,826	0.881	0.843	0.861
256	52,061	51,941	0.872	0.870	0.871

The data dependency is a pair of assignment and reference instructions that pass data through heap areas such as fields or arrays. This is because these instructions cannot be obtained from a simple sequence of method calls or static analysis of method internals. As for fields, when the assignment instruction d writes the value v to the field f of the object obj and then the value is read by the reference instruction u without being overwritten, we consider that the data dependency from d to u exist. Similarly, for arrays, we extract the data dependencies that distinguish the individual elements using read/write subscript instead of the field f. By comparing the written and read values, we can deal with cases where the execution of an assignment instruction was not recorded or where the value was rewritten from a code outside the observation range of the execution.

We take the set of data dependencies obtained from the complete execution trace as the correct answer set, and calculate precision and recall and F-measure of the data dependencies obtained from the execution traces of the two methods, Near-Omniscient Debugging and Time Series Recording. Table 2.2 and Table 2.3 show the accuracy of data dependencies calculated from all benchmarks.

Near-Omniscient Debugging does not record any part of read/write instructions that are executed repeatedly, which causes false positives and false negatives of data dependencies. In particular, fields where a certain range of values are as-

Table 2.3: Accuracy of Data Dependencies Obtained by Time Series Recording (Sum of All Benchmarks).

	Correct	Time Series Recording			
k	Dependency	Dependency	Precision	Recall	F-measure
16	52,061	3,010	1.000	0.058	0.109
32	52,061	3,144	1.000	0.060	0.114
64	52,061	3,938	1.000	0.076	0.141
128	52,061	4,043	1.000	0.078	0.144
256	52,061	4,054	1.000	0.078	0.144

Table 2.4: Accuracy of Data Dependencies Obtained by Near-Omniscient Debugging (Each Benchmark, $k = 16$).

	Correct	Near-Omniscient Debugging			
Benchmark	Dependency	Dependency	Precision	Recall	F-measure
avrora	4,325	3,269	0.904	0.683	0.778
batik	7,054	6,366	0.951	0.858	0.902
fop	13,906	12,143	0.968	0.845	0.902
jython	20,460	15,342	0.849	0.637	0.728
luindex	4,098	3,699	0.952	0.859	0.903
lusearch	2,218	1,784	0.915	0.736	0.816

signed or referenced in many places, such as global variables that represent state transitions of objects, are the cause of many false positives and false negatives. For example, the avrora.arch.legacy.LegacyInterpreter class included in the avrora benchmark is an interpreter implementation that executes a given program. More than 100 methods iteratively manipulate the fields related to the program counter (pc, $nextPC$) in this benchmark. Due to a large number of executions of each assignment and reference instruction, Near-Omniscient Debugging did not preserve the execution order of the instructions sufficiently, leading to a large number of missing data dependencies and incorrectly detecting the execution of different instructions using the same value as data dependencies. However, these false positives were limited to a small number of fields, and the overall precision was more than 0.9, and the recall was around 0.8, which was high even when the recorded data size was limited to 1% comparing with Omniscient Debugging. On the other hand, since Time Series Recording retains only the most recent execution trace, it does not recognize incorrect data dependencies, but it is not suitable for investigating the entire program since it discards most of the data dependencies.

Table 2.4 and Table 2.5 show the detailed results for each benchmark at $k = 16$. The results for other values of k are omitted because the trend for each benchmark remained the same even when the value of k was changed. From these results, we can see that the proposed execution trace reduction method shows high values for all benchmarks, indicating that Near-Omniscient Debugging is a promising execution trace reduction method. The reason why precision and F-measure of lusearch are N/A is that the data dependencies on fields and arrays could not be

Table 2.5: Accuracy of Data Dependencies Obtained by Time Series Recording (Each Benchmark, $k = 16$).

	Correct	Time Series Recording			
Benchmark	Dependency	Dependency	Precision	Recall	F-measure
avrora	4,325	289	1.000	0.067	0.125
batik	7,054	50	1.000	0.007	0.014
fop	13,906	2,149	1.000	0.155	0.268
jython	20,460	137	1.000	0.007	0.013
luindex	4,098	385	1.000	0.094	0.172
lusearch	2,218	0	N/A	0.000	N/A

obtained by the Time Series Recording method.

2.5 Evaluation on Effectiveness for Debugging

In this section, we investigate the usefulness of Near-Omniscient Debugging using Defects4J, a dataset of actual bugs. We investigate two perspectives in the following Research Question.

- RQ1. How is the runtime overhead of Near-Omniscient Debugging?
- RQ2. Are the execution traces recorded by Near-Omniscient Debugging useful for debugging actual bugs?

In the RQ1, we investigate the time and storage cost of applying Near-Omniscient Debugging to an actual bug dataset. We compare the costs of running the tests normally, using Near-Omniscient Debugging and conventional Omniscient Debugging.

In the RQ2, we investigate whether the execution traces reduced by Near-Omniscient Debugging for actual bugs can record the bug-related instructions. We conducted an analysis on the recording rate of bug-related instructions for each project and the difference in the recording success rate when the buffer size of the Near-Omniscient Debugging was changed.

2.5.1 Experimental Settings

We used Defects4J version 2.0.0[1] on OpenJDK Runtime Environment (build 1.8.0_252-8u252-b09-1 18.04-b09) for our analysis.

The dataset includes a collection of bug fixes in open source software projects, containing the source code and unit tests that caused test failure before and after bug fix. Table 2.6 shows the number of bug fixes we used in this case study, whose execution trace is recorded successfully.

Our dataset excludes deprecated bugs[2] because they are not reproducible due to behavioral changes introduced under Java 8. Our dataset also excludes three bug fixes (Lang 43b, Math 13b, and 14b) because their `OutOfMemoryError` bugs

[1] `https://github.com/rjust/defects4j/`, Commit `9349e37`
[2] `https://github.com/rjust/defects4j/tree/v2.0.0`

Table 2.6: The Defects4J Dataset Used for Evaluation.

Name	#Analyzed Bugs	Excluded Bugs(Ids)	Deprecated Bugs(Ids)
Chart	26		
Cli	39		1 (6)
Closure	174		2 (63,93)
Codec	18		
Collections	4		24 (1-24)
Compress	47		
Csv	16		
Gson	18		
JacksonCore	26		
JacksonDatabind	112		
JacksonXml	6		
Jsoup	92	1 (67)	
JxPath	22		
Lang	63	1 (43)	1 (2)
Math	104	2 (13,14)	
Mockito	38		
Time	26		1 (21)
Total	831	4	29

cannot be recorded by our trace recorder, and excludes one bug fix (Jsoup 67b) because the cause of the error is too much execution time, which also cannot be recorded by our trace recorder.

In the RQ1, we measure the cost of recording execution traces of our Near-Omniscient Debugging by comparing it with Omniscient Debugging and normal execution. First, we measure the execution time and storage cost of running one test that causes the bug for each bug ID in the dataset. We ran the recorder component named SELogger in Near-Omniscient Debugging mode and Omniscient Debugging mode and compared the results with normal execution. We also measure the overall cost of the test by running all tests for each benchmark project and identifying the bug ID with the largest number of executed instructions. For this measurement, we used SELogger's mode that only measures the number of executed instructions. We measure the time cost and storage cost of running all the tests with Near-Omniscient Debugging and Omniscient Debugging for each bug ID which records the largest number of executed instructions.

In the RQ2, we investigate whether the execution traces reduced by Near-Omniscient Debugging for actual bugs can record the bug-related instructions. In debugging process, the important thing is whether the execution trace required for debugging is recorded completely. Zeller said that the information required for debugging is the instruction from the location embedded the bug as the defect to the location occurring the bug as the failure [13]. Based on this, we define **Bug-related Instruction**. In concrete, we record and compare the Defects4J's buggy execution and fixed execution using Omniscient Debugging, and define **Bug-related Instruction** as the instruction containing the values, which appears only in the buggy execution and does not appear in the fixed execution.

Following this definition, we conduct an analysis of the complete recording rate of **Bug-related Instructions** by Near-Omniscient Debugging for each project

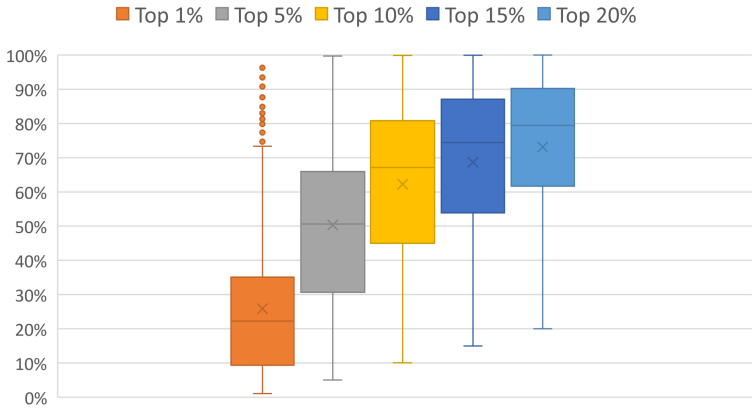

Figure 2.4: Distribution of the Percentage of Execution Trace Size Occupied by Top N% Instructions for Each Bug ID When Instructions are Sorted in Descending Order of Execution Counts.

and the difference in the complete recording rate when the buffer size of Near-Omniscient Debugging is changed. In our experiment, we create the execution trace of Near-Omniscient Debugging from that of Omniscient Debugging to minimize the impact of differences in each execution.

We also create the execution trace which cannot be recorded by Near-Omniscient Debugging by excluding the execution trace of Near-Omniscient Debugging from that of Omniscient Debugging for both the trace of a buggy and fixed execution trace. For these traces, if the fixed execution trace matches or includes the buggy execution trace, we can conclude that all **Bug-related Instructions** are correctly recorded by Near-Omniscient Debugging.

In this evaluation, we do not exclude the instructions whose value changes with each execution, such as time information. These instructions should be excluded from this evaluation, but it is difficult to identify these instructions and the instructions on which they depend. Furthermore, since it is difficult to determine whether the difference in the value in instruction is truly unrelated to the bug, we consider the complete recording to have failed if there is a change in the values in instruction.

We investigated the characteristics of the target of this experiment before conducting the evaluation. In order to discuss how much of the **Bug-related Instruction** could be completely recorded, it is necessary to understand the bias in instruction execution counts in the target bugs. Therefore, in order to check whether a particular instruction is repeatedly executed, we investigated the amount of execution traces occupied by a small number of instructions.

Figure 2.4 shows the distribution of the percentage of the size of execution trace occupied by top N% instructions for each bug ID when instructions are sorted in descending order of execution count. The figure shows that the instructions in the top 20% of the instructions accounted for 80% of the size of execution trace for about half of the bugs. In about three-quarters of the cases, the top 1% of instructions account for more than 10%, indicating that a small number of instructions are repeatedly executed in many bugs.

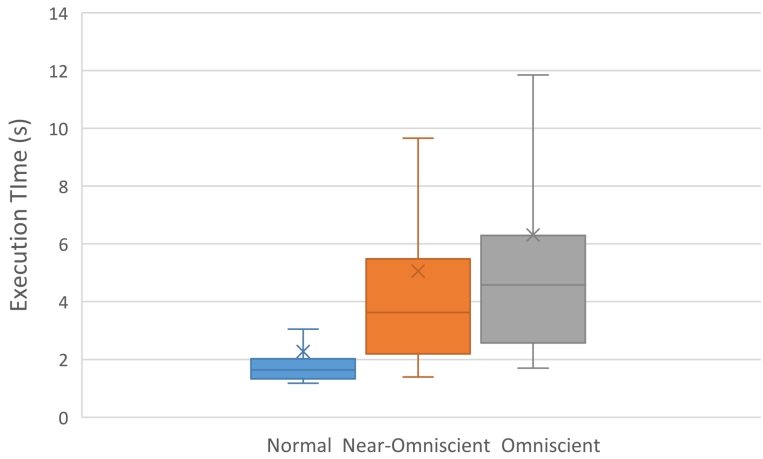

Figure 2.5: The Distribution of Execution Time for Each Bug ID.

On the other hand, there are bugs in which a small number of instructions do not account for the majority of instructions, although the percentage is relatively small. In about a quarter of the cases, even if the top 20% of instructions are excluded, more than 40% of the execution trace remains.

These results indicate that the majority of the bugs are those with a large bias in the number of instructions executed instruction, but there are also a certain number of bugs with a small bias.

2.5.2 RQ1. How is the runtime overhead of Near-Omniscient Debugging?

In RQ1, we analyze the execution cost of the Near-Omniscient Debugging in terms of execution time and size of execution trace, comparing normal execution and Omniscient Debugging.

Figure 2.5 shows the distribution of the execution time for each bug ID. We found that the execution time of Near-Omniscient Debugging was larger than that of Omniscient Debugging when the benchmark execution time and recorded size of execution trace were short. This is because Near-Omniscient Debugging keeps the collected instructions in memory and outputs them when the execution is finished, while Omniscient Debugging outputs the instructions immediately after collection. In other words, when the amount of recorded data is the same, Near-Omniscient Debugging requires more processing and the execution time becomes longer. On the other hand, when the size of the execution trace is large, Near-Omniscient Debugging reduces the output overhead by decreasing the output size of the execution trace, and the execution time becomes shorter.

Then, we describe the size of the execution trace when executing a test in which a bug occurs. Figure 2.6 shows the distribution of the size of execution traces recorded by Near-Omniscient Debugging with buffer sizes varying from 16 to 1024 and Omniscient Debugging. The results show that the Near-Omniscient Debugging significantly reduces the size of execution trace, especially for cases

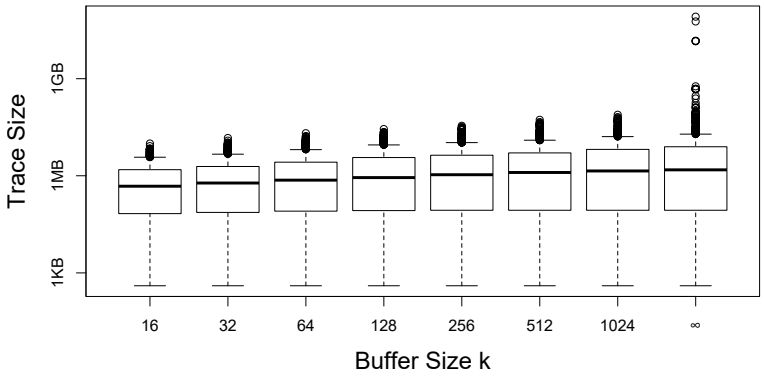

Figure 2.6: The Distribution of the Size of Execution Trace for Each Bug ID at Each Buffer Size.

with particularly large executions that exceed 1GB by Omniscient Debugging. In addition, shown in Figure 2.4, in almost half of the bug IDs, 20% of executed instructions occupy 80% of the size of the execution trace. It means that Near-Omniscient Debugging can significantly reduce the size of execution trace for executions with many repetitions, such as loop statements that are executed more than 10 million times, and reduce the size of the execution trace. From this, we can conclude that the Near-Omniscient Debugging achieves a reduction in the size of execution trace, especially for those repeatedly executed instructions.

Next, we measure the cost of running all tests in projects. Table 2.7 shows the information for bug IDs with the largest size of execution trace recorded by Omniscient Debugging for each project. Note that only the execution of Mockito in this table uses a different version of SELogger [3], since we found a bug related to string recording during the experiment. This bug fix does not affect other executions.

We describe the execution time when all the tests are executed. We can see that the execution time of near Omniscient Debugging is shorter in all the projects than that of Omniscient Debugging. This can be attributed to the fact that many instructions were executed repeatedly, and the overhead of recording the execution was incurred.

We also describe the size of the execution trace. Note that *Math* stopped the execution when it exceeded 10TB. Omniscient Debugging recorded more than 1GB in all but two cases, while Near-Omniscient Debugging recorded less than 1GB in all but three cases. Although the reduction rate varies depending on the project, we found that there was a significant difference in the number of values recorded by Omniscient Debugging compared to Near-Omniscient Debugging. If there is even a single test that contains a large number of iterations, the size of the execution trace will be so large that it may overflow like *Math*, making the application of Omniscient Debugging impractical. On the other hand, Near-Omniscient Debugging can calculate the upper limit in advance from the number of instructions, and the actual recorded size of execution trace is smaller than that of Omniscient Debugging, less than 1GB in most cases, so it is practical enough.

[3]CommitID: cc8ac211d82674f8ea7fdebab59b7e25a38f01a4

We can also discuss the estimation of the size of the execution trace. The correlation coefficient between the size of execution trace recorded by Near-Omniscient Debugging and the number of methods is 0.806, indicating a strong positive correlation. On the other hand, the correlation coefficient between the size of execution trace recorded by Omniscient Debugging and the number of methods is -0.237, indicating a weak negative correlation. These results show that the size of execution trace recorded by Near-Omniscient Debugging method can be estimated in advance if the number of methods that will be executed in the tests of the target project is given.

RQ1. Answer

In the case of executing failed tests, Near-Omniscient Debugging can greatly reduce both the size of execution trace and execution time for a few cases compared with Omniscient Debugging. In the other cases, there was little difference in the size of execution trace and execution time. In addition, we confirmed that the size of the execution trace and the execution time was significantly reduced when all tests were executed. Furthermore, The size of execution trace recorded by Near-Omniscient Debugging has a strong positive correlation with the number of methods in the program. It means that developers can estimate the size of the execution trace before the execution easily.

Table 2.7: The Execution Cost of Bugs with the Largest Execution Trace for Each Project.

				Time (sec.)		Trace Size (GB)	
Project	ID	#Class	#Method	Near-Omniscient(k=64)	Omniscient	Near-Omniscient(k=64)	Omniscient
Chart	6	889	10,454	1,348	2,283	0.2	355.6
Closure	107	2,447	29,874	1,774	2,625	4.6	183.7
Cli	21	125	1,245	63	79	0.02	0.3
Codec	11	115	1,024	1,871	2,460	0.03	471.0
Collections	28	908	8,861	482	767	0.3	27.4
Compress	47	990	7,327	2,451	3,515	0.2	656.0
Csv	16	628	8,258	299	447	0.02	74.5
Gson	14	713	3,186	125	157	0.09	1.7
JacksonCore	25	250	3,376	1,154	1,898	0.2	279.8
JacksonDatabind	56	3,431	19871	998	1,381	1.4	37.3
JacksonXml	6	1,139	13,171	210	236	0.3	0.4
Jsoup	84	565	5,499	123	233	0.1	16.0
JxPath	18	516	5,677	184	527	0.3	15.4
Lang	63	336	4,548	1,687	3,612	0.03	745.7
Math	8	1,744	13,514	78,480	N/A	0.9	>10TB
Mockito	2	1,657	38,728	753	1,233	2.0	37.9
Time	6	517	9,731	1,035	1,366	0.4	214.5

Table 2.8: Complete Recording Rate of **Bug-Related Instructions** at Each Buffer Size.

Buffer Size	Complete Recording Rate
16	38.6%
64	54.9%
256	71.2%
1024	84.2%

Table 2.9: The Number of Bug IDs Recording **Bug-Related Instructions** Completely and with Trace Reduction at Each Buffer Size.

Buffer Size		16	64	256	1024
w/o Difference	w/o Reduction	156	242	402	528
	w/ Reduction(w/ buggy inst.)	165(18)	214(28)	190(26)	172(27)
w/ Difference	w/ Reduction	510	375	239	131

2.5.3 RQ2. Are the execution traces recorded by Near-Omniscient Debugging useful for debugging actual bugs?

In this research question, we describe the complete recording rate of **Bug-Related Instructions** by Near-Omniscient Debugging. We measure whether **Bug-Related Instructions** are completely recorded for each bug ID and buffer size and calculate the complete recording rate.

Table 2.8 shows the percentage of bugs whose **Bug-Related Instructions** are completely recorded for each buffer size. Moreover, Table 2.9 shows the detailed results based on a complete recording of **Bug-Related Instructions** and the execution trace reduction. In half of the bug IDs with a buffer size of 64 and nearly 90% of the bug IDs with a buffer size of 1024, Near-Omniscient Debugging can record **Bug-Related Instructions** completely. Bugs without execution trace reduction mean that all instructions were executed less than or equal to the buffer size. It means that the size of the execution trace of these bugs is the same for both Near-Omniscient Debugging and Omniscient Debugging. The characteristics of Near-Omniscient Debugging is a complete recording of **Bug-Related Instructions** with execution trace reduction. Table 2.9 shows that more than half of the bugs succeed in a complete recording of **Bug-Related Instructions** in the case of execution trace reduction when the buffer size is 1024. The table also shows the number of bug IDs in which the execution trace of **Bug-Related Instructions** is reduced. These numbers don't account for a large percentage of the total numbers of bug IDs with a complete recording of **Bug-Related Instructions** and execution trace reduction. It means that Near-Omniscient Debugging reduces the execution trace of bug-unrelated instructions in many cases when complete recording of **Bug-Related Instructions** is successful with reduction of the size of the execution trace. These results indicate that Near-Omniscient Debugging can record **Bug-Related Instructions** for most of the bugs if the buffer size is above a certain level.

Table 2.10 shows the number of bug IDs whose **Bug-Related Instructions**

Table 2.10: The Number of Bugs Including Incompletely Recorded **Bug-Related Instructions** at Each Buffer Size.

Project	#Bug	16	64	256	1024
Chart	26	6	2	0	0
Cli	39	7	4	0	0
Closure	174	164	156	121	74
Codec	18	5	2	1	1
Collections	4	1	0	0	0
Compress	47	21	14	8	2
Csv	16	2	0	0	0
Gson	18	4	1	1	1
JacksonCore	26	8	3	2	1
JacksonDatabind	112	86	51	15	1
JacksonXml	6	2	2	1	0
Jsoup	92	63	41	18	6
JxPath	22	20	10	9	3
Lang	63	14	5	0	0
Math	104	57	42	33	20
Mockito	38	26	24	23	21
Time	26	24	18	7	1
Total	831	510	375	239	131

are incompletely recorded for each buffer size and each project. We can see that there is a difference in the trend of the rate of completely recorded **Bug-Related Instructions** for each project. In particular, when the buffer size is 256, most of the projects have few bugs with incomplete records, while some benchmarks, such as Closure and Jsoup, have many incomplete records. This is largely due to the characteristics of the projects. For example, in the case of Closure, the internal compilation work is divided into multiple threads, and the execution differs each time depending on the memory and CPU status. Therefore, most of the cases are considered to have failed when the buffer size is less than 256.

We perform a detailed analysis of the cases where Near-Omniscient Debugging failed to complete recording. Figure 2.7 shows the distribution of incomplete recording rate of **Bug-Related Instructions** for each bug IDs at each buffer size. The number of bug IDs that Near-Omniscient Debugging fails complete recording of **Bug-Related Instructions** is given as N in the figure.

As the buffer size increases, the number of bug IDs that failed to complete recording of **Bug-Related Instructions** decreases, and the incomplete recording rate of **Bug-Related Instructions** decreases significantly. In particular, at a buffer size of 256, the third quartile of the failure rate is quite small, at about 10%, indicating that even when Near-Omniscient Debugging fails complete recording, most of **Bug-Related Instructions** are completely recorded.

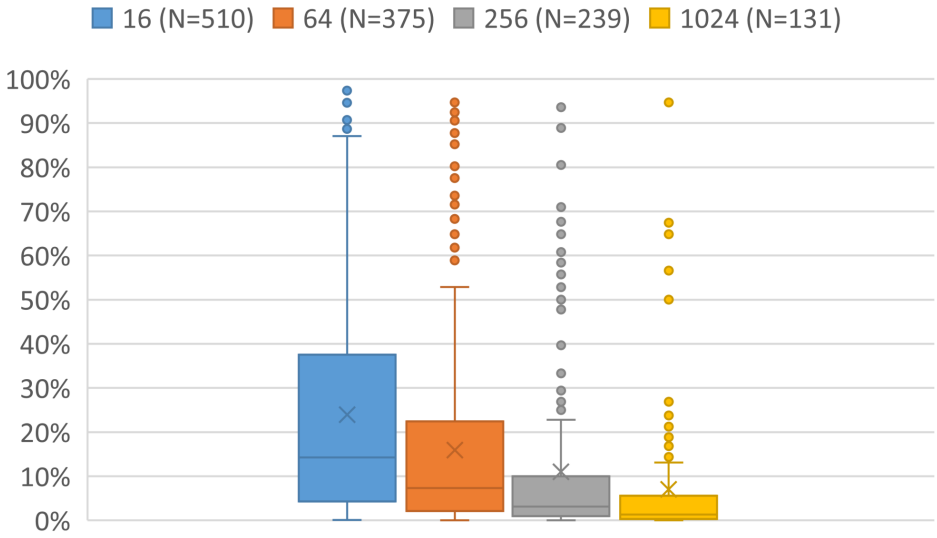

Figure 2.7: The Distribution of the Incomplete Recording Rate of Bug-Related Instructions for Each bug IDs at Each Buffer Size When Complete Recording Fails.

RQ2 Answer

We found that Near-Omniscient Debugging was able to completely record **Bug-Related Instructions** for most of the bugs. In concrete, Near-Omniscient Debugging succeeds in completely recording **Bug-Related Instructions** with 71.2% of the bug IDs when the buffer size is 256. Even in the case where the complete recording of **Bug-Related Instructions** failed, the majority of those were completely recorded. In conclusion, Near-Omniscient Debugging records **Bug-Related Instructions** in the majority of bugs sufficiently, so this method is useful for debugging.

2.6 Discussion

Near-Omniscient Debugging records the latest k times for each instruction, which means there will be some missing values. Since old observed values are missing in many repeated loops, it is not possible to observe whether the process was carried out as expected in the loop when values assigned outside the loop are used in the loop. Therefore, this method cannot reproduce a complete execution, which is a debugging method that is possible when all observed values are completely recorded, or visualize complete data dependencies, as is possible with Omniscient Debugging. It is possible to recalculate missing observed values from the recorded ones, but not all of them can be recalculated. This is a limitation of Near-Omniscient Debugging, which cannot handle complete execution due to partially missing data. However, the evaluation in this chapter has shown that there is little missing information when Near-Omniscient Debugging is actually used.

Therefore, we consider that Near-Omniscient Debugging can be used to support developers in their debugging work when it is applied to partial execution rather than the complete execution.

2.7 Conclusion and Future Work

In this chapter, we proposed Near-Omniscient Debugging to monitor detailed software execution with reducing storage space consumption. Near-Omniscient Debugging takes as input k to specify the number of recorded values for each instruction. It retains actual values for 60% to 74% of instructions and keeps many of the data dependencies on fields and arrays using fewer than 1% of the full execution traces. Near-Omniscient Debugging can also record **Bug-Related Instructions** in the majority of bugs.

We would like to investigate effective logging for textual contents such as strings and exceptions. We also would like to support determining the appropriate buffer size for Near-Omniscient Debugging because this parameter varies greatly depending on the usage of the execution trace recorded by this method, the characteristics of the project, and the size of the storage.

Chapter 3

NOD4J: Near-Omniscient Debugging Tool for Java Using Size-Limited Execution Trace

3.1 Introduction

Debugging is a methodology used to identify defects in the source code by diagnosing its external behavior. For efficient debugging, developers monitor the execution order of instructions and actual values of variables in the source code [32]. Additionally, developers may compare program behavior at the various points of execution where a failure does or does not occur [33, 34]. Interactive visualization tools such as JIVE [45] and break-point debuggers are useful for such analysis; however, developers struggle to use these tools for systems that run on continuous integration and web application servers.

Logging is a common practice that is used to record a program execution as a sequence of messages that report a software's progress and its important data [35]. However, logs recorded in production may not contain sufficient data for debugging because the data to be logged is determined at development time [36]. Therefore, to enable efficient debugging, an automatic method capable of recording a detailed program's execution is needed.

Omniscient debugging [22] is a method that can be used to record all the runtime events during program execution. Although the method enables developers to inspect the state of a program at an arbitrary point in execution, it results in a huge execution trace, and in some cases, grows as fast as 10 MB per second [28]. Developers have difficulty estimating the size of an execution trace prior to execution. Therefore, this implies difficulty in determining what data should be logged to fix bugs in a deployed environment [38].

In Chapter 2, we proposed Near-Omniscient Debugging for Java using size-limited execution traces. Since a full execution trace includes many uninteresting method calls such as utility functions [38], we introduce a parameter k that specifies

the maximum number of recorded values for each instruction. This parameter limits the size of an execution trace for repeatedly executed instructions, while keeping all actual values of the variables associated with instructions that are executed less than k times.

In this chapter, we present a tool NOD4J (Near-Omniscient Debugger for Java), which records and visualizes an execution trace within limited storage space constraints. The tool records local variables and fields used in a Java program execution by Near-Omniscient Debugging proposed in Chapter 2 and annotates the source code with the recorded values. We describe the implementation details and show two usage examples of debugging actual bugs.

In the remainder of the chapter, Section 3.2 explains the background of the tool and Section 3.3 describes its implementation. Then, Section 3.4 describes usage examples and Section 3.5 discusses the limitation of NOD4J. Finally, Section 3.6 concludes the section and describes future work.

3.2 Background

Inspecting the runtime behavior of a program is an important activity for debugging. To achieve this goal, omniscient debugging records all the runtime events such as memory access inside a program. Similarly, Record-and-Replay methods [46] [47] [48] record all the interactions between a program and its external environment. As with omniscient debugging, the approach may result in a huge execution trace [48].

One of the causes for larger execution traces is the repetition of the program instructions. To reduce the size of the execution traces, various compression and sampling methods have been proposed. Wang et al. [39] proposed an effective compression method tailored for execution traces comprising a sequence of memory addresses accessed by a program. Cornelissen et al. [38] reported that execution traces excluding unimportant utility functions retain more information than a trace filtered by a simple sampling algorithm using the same storage space. Hizrel et al. [40] proposed Bursty Tracing, a sampling method that periodically turns monitoring on and off. However, users of those methods cannot estimate the size of a trace prior to program execution. NOD4J enables users to specify the size of an execution trace.

NOD4J is a tool to visualize the values of variables recorded in program execution. Exiting tools, JIVE and Querypoint, provide similar features. JIVE is an interactive execution environment for Eclipse that visualizes a Java program execution at runtime [45]. While JIVE adds useful features to Eclipse debugger, it cannot visualize a program execution outside of the debugger such as continuous integration. Querypoint is a Firefox plugin, which provides critical information for debugging JavaScript programs [49]. This tool provides a program location where a questionable value was assigned. Instead of recording an execution trace, Querypoint executes the program twice; The first execution observes values and the second execution identifies the point where a questionable value was assigned.

Table 3.1: Major Runtime Events Recorded by the Recorder Component.

Event Category	Event Name	Recorded data
Method Execution	Method Entry	Receiver object
	Method Param	Parameter given to the method
	Method Normal Exit	Returned value from the method
	Method Exceptional Exit	Exception thrown by the method
Method Call	Call	Receiver object
	Call Param	Parameter passed to the callee
	Call Return	Returned value from the callee
Local Variable	Local Load	Value read from the variable
	Local Store	Value written to the variable
	Local Increment	Value written to the variable by an increment instruction
Field Access	Get Instance Field	Object whose field is read
	Get Instance Field Result	Value read from the field
	Put Instance Field	Object whose field is written
	Put Instance Field Value	Value written to the field
Array Access	Array Load	Accessed array to read
	Array Load Index	Accessed index to read
	Array Load Result	Value read from the array
	Array Store	Accessed array to write
	Array Store Index	Accessed index to write
	Array Store Value	Value written to the array
	Array Length	Array whose length is referred
	Array Length Result	The length of the array

3.3 NOD4J Overview

NOD4J records a partial execution trace of a Java program and generates an HTML-based view to interactively explore the recorded trace on a web browser. The key idea and recording method of our tool are described in Section 2.3.

In this section, we describe the detailed implementation of NOD4J. Our tool comprises three components: trace recorder, post-processor, and interactive view. The trace recorder component records an execution trace of a Java program in storage. The post-processor component links the recorded trace to the source files of the program. The interactive view shows the source code contents annotated with trace information. The following subsections explain each component in detail.

3.3.1 Trace Recorder

Our recorder component named SELogger (Logger for Software Engineering research) is an extension of the existing trace recorder for REMViewer[43], an omniscient debugging tool. It is implemented as a Java agent working inside a Java Virtual Machine. During program execution, the recorder monitors class loading events and injects logging instructions into the loaded classes using ASM, a Java bytecode manipulation framework.[1] The injected logging instructions are executed as a part of the target program. The standard library classes (e.g., `java`

[1] `https://asm.ow2.io/`

```
 1: void methodD() {
 2:     ...
 3:     // record a Call event of methodE
 4:     this.methodE();
 5:     // record a Call Return event of methodE
 6:     ...
 7: }
 8:
 9: void methodE() {
10:     // Record a Method Entry event of methodE
11:    try {
12:        // Original content of methodE
13:        ...
14
15:        // Record a Method Exit event of methodE
16:    } catch (Throwable e) {
17:        // Record a Method Exceptional Exit event of methodE
18:        throw e;
19:    }
20: }
```

Figure 3.1: An Example of Logging Code Locations for Recording Method Execution and Call Events.

and `javax` packages) are excluded from injection in order to avoid license issues.

Table 3.1 shows a list of major runtime events recorded by the component. To capture inter-procedural control-flow in program execution, the recorder component uses two types of events defined in AspectJ [50]: Method Execution and Method Call. A Method Execution event is recorded when a method body is executed (i.e., on a callee side), while a Method Call event is recorded before a method invocation instruction is executed (i.e., on a caller side). Figure 3.1 shows an example code fragment indicating the locations of logging instructions for `methodE`. In the code fragment, a Call event is recorded at line 3, and then `methodE` is called at line 4. The Call event is followed by a Method Entry event of an actually executed method depending on the type of `this` object; line 10 records a Method Entry event if the method is selected by dynamic binding. When the execution of the method is finished, a Method Normal Exit event representing the location of a return statement is recorded. Finally, a Call Return event is recorded on the caller side. The sequence of events enables us to trace the actual execution path determined at runtime. In addition to the control-flow information, their arguments, return values, and exceptions are also recorded in an execution trace. It is worth noting that a single instruction may be recorded as multiple runtime events; for example, a method execution is represented by a Method Entry event recording a receiver object and a number of Method Param events recording the parameters.

The recorder component also records three types of memory access: local variables, fields, and arrays. The Local Load and Local Store events record actual values read from and written to local variables in a trace. The Local Increment

event represents a pair of Local Load and Local Store performed by an increment instruction (e.g. "`i++`") that frequently appears in a program. Events in the Field Access category record object references in addition to values read from and written to the fields. Similarly, events in the Array Access category record array references and their accessed indices. Although most memory read events just record the same values as their preceding memory write events, the tool records all the Field Access and Array Access events because their values may be updated by library classes whose behavior is not observed by the tool.

An execution trace is a sequence of events $\langle d, t, v \rangle$ where d is a *data ID* to identify an event of a particular instruction, t represents the thread that executed the instruction, and v represents the observed value. A data ID d is assigned by the recorder component during the bytecode instrumentation. When a method is called, the recorder component assigns four IDs to represent the following method execution events.

- A method entry event recording the receiver object (that is, the value of `this`).
- An actual parameter event that records the value of the argument passed to the method.
- A normal exit event. Although the method does not return any value, the event represents a successful termination of the method.
- An exceptional exit event recording an exception if the method is terminated by an exception.

The component also assigns data IDs to every instruction in the method.

Our tool records the latest k events for each data ID d. Then, it takes buffer size k as a parameter, allocates buffers for each data ID, and accumulates the observed events alongside their sequence numbers (i.e., an increasing number representing the order of events). When the program has finished, the tool writes the accumulated data to storage using a shutdown hook function in the Java virtual machine. The maximum trace size is $k \times N$ where N is the number of data elements used by instructions in a program. Our tool with $k = \infty$ is conceptually equivalent to omniscient debugging.

To enable users to investigate how objects are manipulated, our tool translates object references into object IDs composed of their class name and a number. For String and Exception objects, the tool records their textual contents with object IDs for ease of debugging. Our current implementation simply records the textual contents as is. For this reason, in case of long strings, the trace size could still be large. Thus, an effective recording of textual contents will be considered in future work.

3.3.2 Post Processor

The post-processor component links the source code contents of a program to data elements in an execution trace of the program produced by the recorder component. Conceptually, the output of the component is a mapping $\{l \mapsto d\}$, where l is the location of an identifier in the source code and d is a data element in

Table 3.2: Events Linked to Identifiers in Source Code.

Event	Identifiers in source code
Method Param for a parameter `par`	The formal variable name `par` in a method declaration, e.g. "`void m(Type par)`"
Local Load reading a variable `var`	The variable name `var` in an expression
Local Store writing to a variable `var`	The variable name `var` on the left hand side of an assignment expression, e.g. "`var =` e" and "`var +=` e"
Local Increment updating a variable `var`	(1) The variable name `var` on the left hand side of an assignment expression, e.g. "`var = var +` *constant*" (2) The variable `var` in a pre-/post-increment expression: `var++`, `var-`, `++var`, or `-var`
Get Field Result reading from a field `f`	The field name `f` in an expression
Put Filed Value writing a field `f`	The field name `f` on the left hand side of an assignment expression, e.g. "`f =` e" and "`x.f =` e"

Table 3.3: The Pseudo Variables Representing Invisible Values on Source Code.

Event Name	Variable Name
Call Return	_ReturnValue
Array Load Result	_ArrayLoad
Array Store Value	_ArrayStore
Array Length Result	_ArrayLength

a trace. Source-to-trace mapping enables an interactive view to show appropriate data values for each source code location of interest to a user.

Table 3.2 shows the runtime events linked to the source code. The component produces one-to-one mapping for those events because a single identifier in the source code corresponds to a method parameter, a local variable, or a field name recorded as those events. For example, suppose an expression `x=y` is in a source file. The assignment `x=` can either be (1) a Local Store event for a local variable `x` or (2) a Put Field Value event for a field `x` corresponding to the line. If those events are recorded, the identifier is linked to them. The link enables an interactive view to show actual values assigned to `x`. Similarly, the variable `y` is linked to either (1) a Local Load event for a local variable `y` or (2) a Get Field Result event for a field `y`. The identifier is linked to those events so that the interactive view can show actual values read from `y`. To link the events and identifiers, the component builds a syntax tree for each source file, extracts identifiers from the tree, and then identifies their corresponding events in the trace. In the last step, the tool searches local variable access and field access events using the source file name, the line number of the expression, and the identifier names such as `x` and `y`. The search depends on the heuristics as shown in Table 3.2 instead of a deeper semantic analysis of the source code. This is because such a semantic analysis requires the whole program and libraries to take class inheritance and static import mechanisms into account.

Some runtime events have no corresponding tokens in the source code. For example, the return value of a method call may be discarded without an assignment operator. To visualize such invisible events in a trace, the post-processor translates them into pseudo variables as shown in Table 3.3. The pseudo variables are also included in the source-to-trace mapping.

```
3  public class Main {
4      public static void main(String[] args) {
5          methodA();
6      }
7      private static int methodA() {
8          int var = 127;
9          do {
10             if (var % 2 == 0) {
11                 var = var / 2;
12             } else {
13                 var = (var + 1) / 2;
14             }
15         } while (var > 10);
16
17
18 }
```

64
32
16
8

Figure 3.2: Trace View.

```
3  public class Main {
4      public static void main(String[] args) {
5          methodA();
6      }
7      private static int methodA() {
8          int var = 127;
9          do {
10             if (var % 2 == 0) {
11                 var = var / 2;
12             } else {
13                 var = (var + 1) / 2;
14             }
15         } while (var > 10);
16
17
18 }
```

32
16
8

Figure 3.3: Trace View with Filtering.

3.3.3 Interactive View

The interactive view component is an HTML-based view that works on a web browser. Using the source-to-trace mapping produced by the post-processor component, each tab displays a source file, whose variables are highlighted. Directly associating variables with their values and highlighting them allows developers to debug intuitively. By hovering a mouse cursor on a highlighted variable, the actual values of the variable recorded in the trace are displayed. For example, Figure 3.2 shows a screenshot of an interactive view displaying the values of `var` in ascending order by time.

In general, when analyzing the cause of a failure, instead of inspecting the entire execution, the analysis focuses only on the interval related to the defect. To debug in this situation, this interactive view can filter values by specifying a time interval. For example, in Figure 3.2, a user can click on the arrows shown on the right side of each value. A click on the left arrow specifies a start point of an interval, while a click on the right arrow specifies the end point. Figure 3.3 shows the interactive view displaying observed values after assigning 32 to `var` at line 15. If no values are recorded for a variable during the specified time interval, the highlighting for that variable is turned off. In Figure 3.3, the highlighting of `var` at line 13 disappeared as a result of filtering. It should be noted that interactive views share a single filter within multiple tabs. A user can investigate an inter-procedural data-flow by selecting an arbitrary pair of source code locations in a program.

The interactive view also provides a textual representation of runtime events linked to a source file. The following example shows the textual representation for the example program.

```
...
ID:16, Line:5, Variable:_ReturnValue, Seqnum:45, Data:8
ID:28, Line:10, Variable:var, Seqnum:8, Data:127
```

...

Each line shows five attributes of an event: data ID d (`ID`), line number (`Line`), variable name (`Variable`), sequence number (`Seqnum`), and recorded data value (`Data`). On the textual representation, we can perform a keyword search on the recorded values. We can also refer to the invisible values in a table. For example, we cannot check a returned value from the method call at line 5 because the value is not assigned to any variable. In the above example, we can check the value as a pseudo variable `_ReturnValue`, which means the method returned value on the line. The interactive view uses a subset of events recorded by the recorder component. For example, thread ID is available but omitted from the view. This is because a naive visualization of all the details of runtime events may be too complicated for users.

3.3.4 Usage

The tool is available on GitHub. The main repository[2], named `nod4j`, includes the binary files of the tool and the source code of the post-processor and interactive view components. The following command builds the tool from source code using Maven. The `nod4j.jar` file is created in `target` directory.

```
git clone https://github.com/k-shimari/nod4j.git
cd nod4j
mvn package
```

The source code of the recorder component is separated in another repository[3] because the recorder may be used for other research. The tool is dependent on the following tools and libraries. The version numbers show the environment of the authors on Windows 10.

- Java(TM) SE Runtime Environment (build 1.8.0_241-b07)
- Apache Maven (3.6.3)
- Node.js (v12.16.1 LTS)
- npm (6.14.4)

To explain the usage of our tool, we performed a debugging session of a small program. This program is included in the `sample/demo/for_build` directory of the `nod4j` repository. The program implements a method to select the maximum number from the three given numbers as follows:

```
public class Main {
    public static int getMax(int num1, int num2, int num3) {
        // return the maximum number of three arguments
    }
}
```

To test the method, the file `getMaxTest.java` contains the following test cases that provide parameters 10, 20, and 30 in different orders to the target method.

[2] `https://github.com/k-shimari/nod4j`
[3] `https://github.com/takashi-ishio/selogger`

```
10:         @Test
11:         public void getMaxTest1() {
12:             assertEquals(30, Main.getMax(30, 10, 20));
13:             assertEquals(30, Main.getMax(30, 20, 10));
14:             assertEquals(30, Main.getMax(20, 10, 30));
15:             assertEquals(30, Main.getMax(20, 30, 10));
16:             assertEquals(30, Main.getMax(10, 20, 30));
17:             assertEquals(30, Main.getMax(10, 30, 20));
18:         }
```

The following command executes the test cases of the sample program.

```
cd sample/demo/for_build/
mvn test
```

The command reports a failure as follows:

```
java.lang.AssertionError: expected:<30> but was:<20>
    at testsample.getMaxTest.getMaxTest1(getMaxTest.java:17)
```

The last test case at line 17 fails and a log message for the test case shows that the expected return value is 30 but the actual return value is 20.

To debug the problem using our tool, the `pom.xml` file for the program specifies the following argument for a Java VM executing the test case.

```
-javaagent:"../../../selogger-\0.2.3.jar=output=../selogger,size=64,
e=junit/framework/,e=org/junit/,e=org/apache/maven,e=org/hamcrest"
```

The `-javaagent` option points to the recorder component called `selogger-0.2.3.jar` located in the top directory of the main repository. The remaining text is the parameters for recording. The `output=../selogger` parameter specifies an output directory. The `size=64` parameters specify the buffer size $k = 64$. The "`e=...`" parameters specify package names to be excluded from the execution trace. In this configuration, the recorder component excludes the Maven and JUnit classes. The execution trace is stored in the `sample/demo/selogger` directory.

To use the interactive view, we need to link the source code to the execution trace.

```
java -jar nod4j.jar sample/demo/for_build sample/demo/selogger
  src/main/frontend/src/assets/project/demo
```

The three arguments specify a source code directory, an execution trace directory, and an output directory, respectively.

The command produces two json files, `fileinfo.json` and `varinfo.json`, in the specified output directory. We put the json files in `src/main/frontend /src/assets/project/demo` so that the interactive view can read the json files. Then, we can build HTML contents from the json files and start a server for the contents. To build HTML contents, we execute the following commands in the `src/main/frontend` directory.

```
npm install
npm run build
npm run server
```

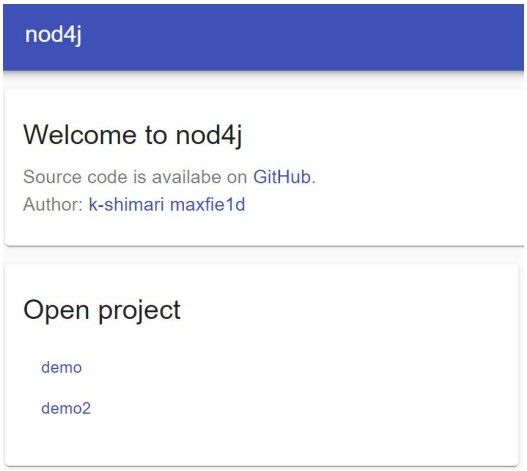

Figure 3.4: The Welcome Page of NOD4J.

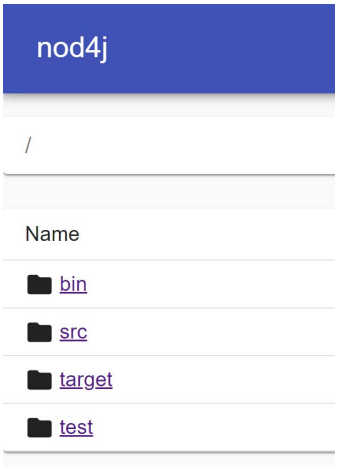

Figure 3.5: The Browser Page of NOD4J.

The `npm install` command is for installing dependencies, so this command needs to be run only once. The `npm run build` command executes the post-processor component. The `npm run server` command starts a web server for the interactive view. A web browser can access the view through a local address `localhost:8070`.

Figure 3.4 shows the welcome page of the interactive view. NOD4J checks subdirectories in `src/main/frontend/src/assets/project/demo` and automatically shows the names in the project list. Then, by clicking the project name and traversing the directory shown in Figure 3.5, we can access the file we want to see the execution.

To analyze the failure of the example program, open the file `src/sample/Main.java`. Figure 3.6 shows the view of the target method in the file. All test cases return a value on the variable `max` at line 26. Then, we can confirm that the sixth return value, 20, caused the test failure. To visualize the computation for this invalid return value, we filter the execution with a time interval from the initialization of `num1` at line 8 by clicking on the left arrow of the sixth value of `num1` at line 8.

Figure 3.7 shows the filtering result. The highlighted variables show that the `max` was incorrectly assigned at line 12. We can fix this bug by changing the variable `num1` to `num2` at line 11.

In this example, the target method includes a loop at lines 23–25 that is irrelevant to the test cases. Although the loop results in a large number of steps in a full execution trace, our tool excludes those steps from the trace.

3.4 Usage Examples

We illustrate two actual debugging sessions using the tool. We use bugs in the Defects4J benchmark version 1.5.0[4] executed on Java(TM) SE Runtime Environment (build 1.7.0_80-b15). We chose Lang 2b and Math 59b as examples, because

[4] `https://github.com/rjust/defects4j/`, Commit `bd32d9642e12`

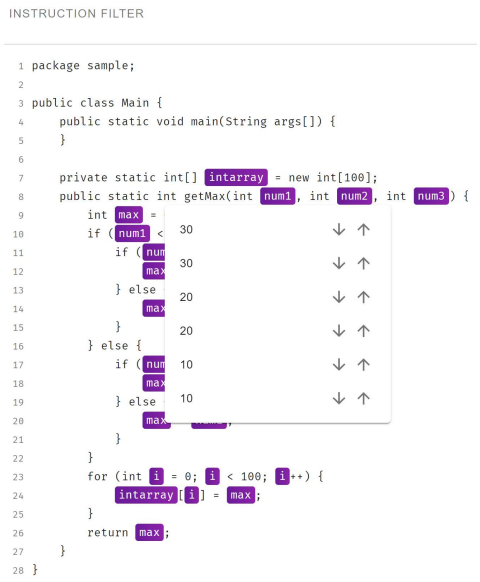

Figure 3.6: The Interactive View for the Current Scenario.

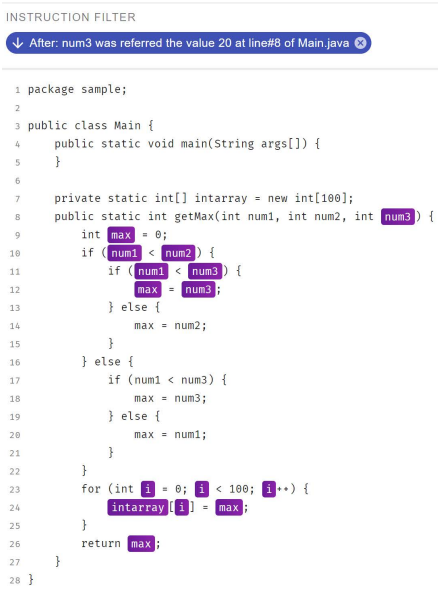

Figure 3.7: The Result of the Filtering.

the former resulted in a relatively smaller execution trace among the bugs and the latter resulted in a larger execution trace. We recorded the execution trace of a failed test method for each bug. When more than one test method failed, we ran only the top one shown in the result of the `detects4j info` command. The traces record only the behavior of the source code of the project. In other words, the recorded traces include method calls to libraries (e.g., `Assertion` class in JUnit) in the target unit tests but do not include the internal behavior of the libraries.

3.4.1 Use Case 1: Lang 2b

The first example is Lang 2b, a bug of the `LocaleUtils.toLocale()` method. The method takes a `String` object representing a locale name and returns a `Locale` object corresponding to the name if the name is in the correct format. The method is tested using a method named `testParse AllLocales()` in the file `src/test/java/org/apache/commons/lang3/LocaleUtilsTest.java`. The test method fails and provides the following log messages to the standard output:

```
Should not have parsed: ja_JP_JP_#u-ca-japanese
Should not have parsed: th_TH_TH_#u-nu-thai
```

To debug this problem, we obtained an execution trace of the failed test method using our trace recorder with $k = 64$. The resulting execution trace completely recorded the behavior of 262 out of 418 instructions (62.7%). The other instructions were executed more than 64 times. The size of the trace is 167 KB, which is 52.2% of its full execution trace (348 KB).

Figure 3.8 shows the test method `testParseAllLocales()` in the interactive view. The test method obtains all available locales and passes their names to

```
545 @Test
546 public void testParseAllLocales() {
547     Locale[] locales = Locale.getAvailableLocales();
548     int failures = 0;
549     for (Locale l : locales) {
550         // Check if it's possible to recreate the Locale using just the standard constructor
551         Locale locale = new Locale(l.getLanguage(), l.getCountry(), l.getVariant());
552         if (l.equals(locale)) { // it is possible for LocaleUtils.toLocale to handle these Locales
553             String str = l.toString();
554             // Look for the script/extension suffix
555             int suff = str.indexOf("_#");
556             if (suff == - 1) {
557                 suff = str.indexOf("#");
558             }
559             if (suff >= 0) { // we have a suffix
560                 try {
561                     LocaleUtils.toLocale(str); // shouuld cause IAE
562                     System.out.println("Should not have parsed: " + str);
563                     failures++;
564                     continue; // try next Locale
565                 } catch (IllegalArgumentException iae) {
566                     // expected; try without suffix
567                     str = str.substring(0, suff);
568                 }
569             }
570             Locale loc = LocaleUtils.toLocale(str);
571             if (!l.equals(loc)) {
572                 System.out.println("Failed to parse: " + str);
573                 failures++;
574             }
575         }
576     }
577     if (failures > 0) {
578         fail("Failed "+failures+" test(s)");
579     }
580 }
```

target locales in this test

if string of locale contains character “#”

output java.lang.AssertionError

Figure 3.8: A Test Method for `LocaleUtils.toLocale` Method in Lang 2b.

the method under test. If the locale name has a suffix (“#”), the method under test should throw an exception. Otherwise, the method should return a `Locale` object corresponding to the locale name. The test method counts the number of occurrences of incorrect behavior using a variable `failures`. The test method failed because the `failures` was greater than zero.

The interactive view provides the actual values of the `str` variable as shown in Figure 3.9. From this information, we can confirm that this test failed because the method under test did not throw `IllegalArgumentException` for locale names including “#”. Figure 3.8 also shows that lines 572 through 573 were not executed because the variables on the lines are not highlighted.

Then, we open the file that includes the target method `toLocale()` to analyze the failure in detail. The first half of the method is shown in Figure 3.10, the

```
559             if (suff >= 0) { // we have a suffix
560                 try {
561                     LocaleUtils.toLocale(str); // shouuld cause IAE
562                     System.out.println("Should not have parsed: " + str);
563                     failures++;
564                     continue; // try next Lo
565                 } catch (IllegalArgumentExce
566                     // expected; try without
567                     str = str.substring(0, suff);
568                 }
569             }
```

java.lang.String@55f23ce:"ja_JP_JP_#u-ca-japanese"

java.lang.String@2a80364e:"th_TH_TH_#u-nu-thai"

Figure 3.9: Actual Parameter Values that Induced a Failure in Lang 2b.

```
88      public static Locale toLocale(final String str) {
89          if (str == nul
90              return null
91          }
92          final int len = str.length();
93          if (len < 2) {
94              throw new IllegalArgumentException("Invalid locale format: " + str);
95          }
96          final char ch0 = str.charAt(0);
97          if (ch0 == '_') {
98              if (len < 3) {
99                  throw new IllegalArgumentException("Invalid locale format: " + str);
100             }
101             final char ch1 = str.charAt(1);
102             final char ch2 = str.charAt(2);
103             if (!Character.isUpperCase(ch1) || !Character.isUpperCase(ch2)) {
104                 throw new IllegalArgumentException("Invalid locale format: " + str);
105             }
106             if (len == 3) {
107                 return new Locale("", str.substring(1, 3));
108             }
109             if (len < 5) {
110                 throw new IllegalArgumentException("Invalid locale format: " + str);
111             }
112             if (str.charAt(3) != '_') {
113                 throw new IllegalArgumentException("Invalid locale format: " + str);
114             }
115             return new Locale("", str.substring(1, 3), str.substring(4));
```

java.lang.String@2a80364e:"th_TH_TH_#u-nu-thai"

Figure 3.10: A Filtering Result of the `toLocale` Method in Lang 2b (The First Half of the Method).

```
116         } else {
117             final char ch1 = str.charAt(1);
118             if (!Character.isLowerCase(ch0) || !Character.isLowerCase(ch1)) {
119                 throw new IllegalArgumentException("Invalid locale format: " + str);
120             }
121             if (len == 2) {
122                 return new Locale(str);
123             }
124             if (len < 5) {
125                 throw new IllegalArgumentException("Invalid locale format: " + str);
126             }
127             if (str.charAt(2) != '_') {
128                 throw new IllegalArgumentException("Invalid locale format: " + str);
129             }
130             final char ch3 = str.charAt(3);
131             if (ch3 == '_') {
132                 return new Locale(str.substring(0, 2), "", str.substring(4));
133             }
134             final char ch4 = str.charAt(4);
135             if (!Character.isUpperCase(ch3) || !Character.isUpperCase(ch4)) {
136                 throw new IllegalArgumentException("Invalid locale format: " + str);
137             }
138             if (len == 5) {
139                 return new Locale(str.substring(0, 2), str.substring(3, 5));
140             }
141             if (len < 7) {
142                 throw new IllegalArgumentException("Invalid locale format: " + str);
143             }
144             if (str.charAt(5) != '_') {
145                 throw new IllegalArgumentException("Invalid locale format: " + str);
146             }
147             return new Locale(str.substring(0, 2), str.substring(3, 5), str.substring(6));
148         }
```

Figure 3.11: A Filtering Result of the `toLocale` Method in Lang 2b (The Second Half of the Method).

second half is in Figure 3.11, respectively. While the method is executed several times, we are interested in only a failed execution whose argument includes "#". Hence, we filter the interesting behavior by clicking on a down arrow on the right side of a string value "`th_TH_TH_#u-nu-thai`" shown in Figure 3.10. This view shows a partial trace recorded after the value was observed at the beginning of the method. The highlighted variables show the execution flow at that time. In Figure 3.10, there is no highlighted variable in the `if` statement's body from line 98 to 115. However, there are many highlighted variables from line 117 to 147 in Figure3.11. Then, we can see that this method call executes `else` statement's body and returns on line 147. At line 147, we can find that the method passed a result of a method call "`str.substring(6)`" to a constructor of `Locale` class. The `if` statements do not check "#" at all in the method. This is the cause of the bug. In the fixed version of the program, the following `if` block is inserted into the method at line 92.

```
if (str.contains("#")) {
 throw new IllegalArgumentException("Invalid locale format: " + str);
}
```

In this debugging session, the actual values of variables recorded in the trace are effective in investigating the behavior of the failed test. While the trace does not record the complete execution, the trace still maintains variable values observed for the corner case.

3.4.2 Use Case 2: Math 59b

The second example is Math 59b. The method `FastMath.max()` compares two `float` values and returns a greater one. The method is tested by the test method `testMinMaxFloat()` in the source file `src/test/java/org/ apache/commons/ math/util/FastMathTest.java`. This test method calls the `FastMath.max()` method with various parameters and compares the results with `Math.max()` as follows.

```
 78: public void testMinMaxFloat(){
 79:  float[][] pairs = {
 80:      { -50.0f, 50.0f },
        ...
 89:  };
 90:  for(float[] pair : pairs){
       ...
 99:   Assert.assertEquals("max(" + pair[0] + "," + pair[1] + ")",
100:                   Math.max(pair[0],pair[1]),
101:                   FastMath.max(pair[0],pair[1]),
102:                   Math.Utils.EPSILON);
103:   Assert.assertEquals("max(" + pair[1] + "," + pair[0] + ")",
104:                   Math.max(pair[1],pair[0]),
105:                   FastMath.max(pair[1],pair[0]),
106:                   Math.Utils.EPSILON);
107:  }
108: }
```

The test method fails and produces the following message.

```
3481    public static float max(final float a, final float b) {
3482        return (a <= b) ? b : (Float.isNaN(a + b) ? Float.NaN : b);
3483    }
3484                                                    -50.0
```

Figure 3.12: Failure Induced Method at Math 59b.

```
java.lang.AssertionError: max(50.0, -50.0) expected:<50.0>
but was:<-50.0> at org.apache.commons.math.util.FastMathTest
.testMinMaxFloat (FastMathTest.java:103)
```

The method under test returned an incorrect value -50.0 for the first value (`-50.0f, 50.0f`), written in line 80.

We recorded an execution trace of the test method with $k = 64$. It resulted in 4.0 MB. 3,841 out of 7,356 instructions (47.8%) were executed more than 64 times; the full execution trace of the test method resulted in 12.4 GB. Our tool recorded only 0.03% of runtime events. The cause of the significant reduction is the setting up of the method defined as follows.

```
@Before
public void setUp() {
  field = new DfpField(40);
  generator = new MersenneTwister(6176597458463500194l);
}
```

Before the unit test is executed, the setup method is executed. The method is short but executes a large number of methods; all the runtime events excluded from our trace belong to the method. The recorded trace includes the complete execution of the test method.

The test targets are `FastMath.min()` and `FastMath.max()`, which return the minimum and maximum values of the two values in the file `src/main/java/org/apache/commons/math/util/FastMath.java`. From the error message, we have to check the execution trace of `FastMath.max()`. Figure 3.12 shows a view of the method. We can easily see the actual values used in the two method calls. Then, we can easily find that we should fix the value of the last variable b to a at line 3,482. This case also shows that a complete execution trace is not always needed for debugging.

3.5 Discussion

There are several cases where this visualization tool may not be able to debug. This tool only collects IDs for objects, so it is not possible to check the specific values recorded in objects. In addition, since this method cannot collect values in external libraries, it is not possible to check and visualize the specific values of variables in external libraries.

We have evaluated this visualization tool only from the perspective of whether the values required for debugging are visualized or not. Since we have not confirmed whether this tool is actually helpful for developers, it is necessary to conduct a qualitative evaluation by developers.

3.6 Conclusion

NOD4J monitors and visualizes detailed software behavior with reducing storage space consumption. The tool is suitable for monitoring remote program execution, such as testing on continuous integration servers, and visualizing the behavior of test failures on a web browser. As illustrated in the usage examples, our tool can debug defects using incomplete execution traces, while it can record complete execution traces for many bugs.

In future work, we would like to investigate effective logging for textual content such as strings and exceptions. Another future research direction is the development of automated debugging methods utilizing near-omniscient execution traces. We also would like to improve the interactive view page so that it can provide an understandable view for large values of k by visualizing the sequence of the values of variables. Finally, as part of our future work, we will ensure that NOD4J responds dynamically to added/removed execution traces.

Chapter 4

Effective Test Case Selection Based on Similarity of Runtime Information for Dependency Update

4.1 Introduction

Regression testing is performed to check the changes in behavior before and after a software change. The cost of regression testing is a significant problem in large-scale software because developers have to repeatedly perform regression testing even when the software is not directly changed. For example, regression testing is necessary for compatibility verification when developers update dependencies used by the software. Chen et al. [51] reported various incompatibilities of JDK and Java libraries detected by test cases in OSS projects.

In this chapter, we report our regression testing activity for updating the dependencies of an enterprise thermodynamic and fluid mechanic simulator. The users of this simulator are engineers of the company who design new products. The simulator accepts various parameters specifying a structure of an electromechanical product and simulates the physical behavior; the users can estimate the efficiency of the structure without constructing an actual prototype. This simulator received more than tens of millions of simulation requests from users in the past five years. The simulator is now recognized as one of the most important simulators for their product development.

The software developers in the company periodically update the simulator and its dependencies and then perform regression testing. For effective regression testing, the simulator automatically records all the users' requests and the simulation results in the storage. When the developers update the simulator, they execute some simulations again and compare the new results with the recorded results. Instead of all the recorded requests, the software developers selected a small subset of test cases that cover available simulation components because it is impractical to execute all test cases due to a limited time and budget.

Although the software developers believed their strategy was sufficient, they failed to detect an incompatibility between JDK versions. They updated the JDK version from Oracle JDK 8 to AdoptOpenJDK 11 and performed regression testing as usual. However, a simulator failure occurred a few months after the regression testing. The cause was a change in the behavior of the Java standard library, which prevented several simulations from running properly. Even though the test cases have been selected to cover simulation components, they did not cover a corner case in a wide variety of behavior of the simulation components. Although test case selection is a popular topic in the software testing community, we could not identify a suitable method to select a small number of effective test cases (at most 100 that could be executed within a day) from a large number of test cases.

To perform effective regression testing for updating dependencies of the simulator, we develop a method to select test cases that result in similar executions using the runtime information of test cases. The method records a lightweight execution trace for each test case. We execute a clustering algorithm to identify similar execution traces and select representative test cases from each of the clusters. The contributions of this chapter are as follows.

- We have developed a method for identifying similar executions of test cases for dependency updates.
- We showed that the clustering method based on runtime information of test cases could be used to select test cases that improve coverage.
- The software developers recognized the selected test cases as effective because they tested a wider range of parameters than their manually selected test cases that have been used for the simulator.
- We showed that the test cases selected by our method could be used to update the dependency in an enterprise simulator.

In the remainder of the chapter, Section 4.2 describes the background of the chapter, and Section 4.3 describes our test selection method. Then, Section 4.4 describes the case study, and Section 4.5 applies our method to an enterprise simulator. Finally, Section 4.6 concludes the chapter and describes future work.

4.2 Background

4.2.1 Dependency Compatibility

In modern software development, dependencies are indispensable and developers should update their dependencies as soon as possible. For example, if there is a critical vulnerability, such as the recent one that allows a remote third party to execute arbitrary code in log4j[1], developers need to update the dependency quickly. However, developers have to deal with the compatibility issue of dependencies. Mostafa et al. [52] reported that backward compatibility of Java software libraries was not maintained in 76.5% of the new versions of libraries. Gyori et al. [53] conducted a study on client-library breakages. This study showed that when updating libraries, 15.0% of compile failures and 11.2% of test failures occur for

[1] https://logging.apache.org/log4j/2.x/security.html

a given library version. This study also showed examples of semantic versioning not working, indicating the need for regression testing. These studies show the importance of regression testing for compatibility.

The incompatibility problems also exist in Java standard libraries. According to Oracle, there are a number of incompatibilities in JDK 11 release[2]. Developers should perform regression tests when updating their JDK version to ensure that the system behavior is the same before and after the update. For example, Java's split method has undergone the following specification changes with Java version updates[3].

Listing 4.1: A backward incompatibility between Java 7 and Java 8.

```
"helloworld".split("")
(OpenJDK 7) : [, h, e, l, l, o, w, o, r, l, d]
(OpenJDK 8 and 11) : [h, e, l, l, o, w, o, r, l, d]
```

In this example, there is a concern that the number of elements in the array will change when JDK is updated, resulting in unexpected behavior such as **ArrayIndexOutOfBoundsException**. It is difficult to identify all such incompatible changes in the release notes when updating the dependency version. Therefore, it is necessary to check whether the dependencies used in the software work properly through regression testing.

4.2.2 Test Case Selection

Regression testing is crucial to ensure that the behavior of the software is the same before and after a software change. Since regression testing is performed repeatedly, it is important to select appropriate test cases to reduce the cost.

Gligoric et al. [24] reported the study of manual test selection in practice and a comparison of manual and automated test selection. The results showed that manual test selection chose more tests than automated selection 73% of the time and chose fewer tests 27% of the time. This means that manual test selection may miss bugs or waste time. In our case, the software developers indeed missed a test case as described in Section 4.1. We need an automated test selection method for regression testing.

Many existing test case selection methods using runtime information have been proposed. Rothermel et al. [26] proposed a method for test case selection using runtime information, which focuses on a source code change. They create control flow graphs of the source code before and after a change and find the modified nodes from the two graphs using a depth-first search. This method considers test cases that pass through these nodes as high risk and selects these test cases. Zhang et al. [27] extended this test selection method. They made vectors of each method call instruction at test executions and calculated the distance between them for clustering. Then, this method executes representative test cases for each cluster on the updated source code and decides whether or not to execute the tests in the cluster by looking at the test results. These methods use runtime information for test case selection, but they also use source code change information for test case selection. However, since the software itself does not change when the dependencies

[2] https://docs.oracle.com/en/java/javase/11/migrate/index.html
[3] https://bugs.openjdk.java.net/browse/JDK-8043324

are updated, we cannot simply apply these existing methods for test case selection of updating dependencies.

Adithya et al. [30] proposed a system that performs data-driven test minimization. This system predicts the test outcomes based on the commit information and the correlation with previous test outcomes to save the test execution time. This technique can reduce the test time by about a factor of five while maintaining 99.99% accuracy of test results for a large-scale service. Machalica et al. [29] proposed a new predictive test selection strategy. This method uses machine learning techniques to learn historical test outcomes. This method can capture more than 95% of individual test failures and more than 99.9% of faulty code, reducing the total infrastructure cost of testing code changes by a factor of two. While these methods are useful, they cannot be applied to our simulator due to their large amount of costs since they require keeping a history of tests and their results, and they require running large test cases for a number of commits.

Gligoric et al. [54] proposed a test case selection technique that can integrate well with testing frameworks. This technique tracks dynamic dependencies of tests on files. When developers modify test files, this technique detects the affected tests from the dependencies and selects these test cases. It reduces the end-to-end testing time by 32% on average compared to executing all tests. The effectiveness is based on the assumption that a small fraction of files is modified at each revision. As a dependency update may change many files at the same time, we could not employ the technique.

4.3 Our Test Selection Method

Our method selects a small number of test cases using runtime information. The objective of this method is to select test cases for updating dependencies in the enterprise thermodynamics and fluid mechanics simulator. The simulator automatically records all the users' requests and the simulation results in the storage for regression testing. The software developers can use these test scripts, but it costs a lot of resources to execute all test cases. The simulator developer would like to select test cases with a coverage close to that of running all the tests, and then select test cases with different execution characteristics, such as a different number of loops with the same coverage. Then, our method selects the combination of test cases that achieves high coverage and diverse execution times from these test scripts, which can cover a wide range of parameters.

Our method consists of two steps. Figure 4.1 shows our method overview. In STEP Ⅰ, we collect runtime information by executing test scripts with an execution trace recorder. In STEP Ⅱ, we classify test cases based on the similarity of their runtime information and select their representative test cases.

4.3.1 STEP I: Collect Runtime Information

At first, we collect runtime information for each test case. To collect the runtime information, we use SELogger version 0.2.3, which is a recorder component of NOD4J described in Chapter 3. SELogger is a Java agent working inside Java Virtual Machine. It records Java bytecode instructions such as method execution and variable access. Detailed implementation is described in Chapter 3.3. By collecting runtime information in the unit of instructions, the detailed behavior of

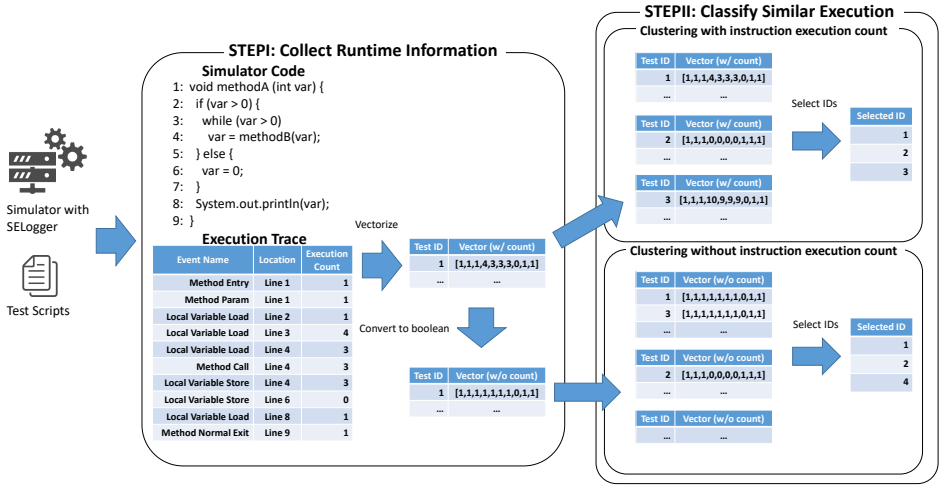

Figure 4.1: An Overview of Our Method.

each test case can be recorded. We use SELogger *freq* mode, which assigns IDs to each of the instructions and records their execution counts. This mode is a lightweight analysis that does not record details of events such as variable values at runtime.

To extract only relevant runtime events to the simulator, we introduced a filtering feature into SELogger. This is because the target simulator is a web application running on Tomcat. While the original version of SELogger records all runtime events including the behavior of Tomcat, our version separately collects only runtime events of the simulator. For each test case, an execution trace is separately collected.

Figure 4.1 shows an example of collecting runtime information. The executed instructions, consisting of runtime events and their location, and their execution counts are recorded as an execution trace for each test script. In this example, the loop from line 3 to line 4 is executed three times, so some events are repeatedly executed, such as *Local Variable Load* event, which means the value is read from the variable, at line 4. From execution traces, We create two types of vectors for comparison: one vectorizes the instruction execution count as is, and the other vectorizes the instruction execution count as a binary value of 0 or 1. In clustering without instruction execution count, clustering is performed based on only coverage. In clustering with instruction execution count, the executed count is used in addition to the coverage so that the execution characteristics such as execution time can be reflected in the vector.

4.3.2 STEP II: Classify Similar Execution

We classify similar executions and select the representative test cases. In both methods described in STEP I, we classify similar vectors using the K-means clustering method, respectively. Since a large number of test cases are recorded in the simulator, scalability is required. Therefore, we adopted the K-means method, which is less computational complexity than hierarchical clustering. This method

Table 4.1: Subsystem of the Target Simulator Under Test.

Metric	Count
#Class	61
#Methods	509
#Lines	21,527

enables fast clustering of large vector data. In the implementation, we use euclidean distance when measuring between vectors. Finally, we select the oldest test case from each cluster as the representative test cases because they are the first test cases that had triggered new behaviors of the simulator.

4.4 Case Study

We conduct a case study to verify the usefulness of our method for updating dependencies. The simulator developer would like to select test cases with a coverage close to that of running all the tests, and then select test cases with different execution characteristics, such as a different number of loops with the same coverage. To evaluate the usefulness of our method in these aspects, this case study compares our method with a method currently used in companies and a random selection method. The research questions of this study are shown below.

- RQ1: Does our test selection method provide better coverage than the existing method?
- RQ2: How diverse is the execution time?
- RQ3: Can our method classify an existing bug-related test case?

Since this simulator has tens of millions of test cases, we should run all the test cases and apply our method. However, it is difficult to execute them in a short period of time due to resource constraints. Instead, we collect ten days of test cases for the case study, with a maximum of one thousand test cases per day, and then select representative test cases from the collected cases. The number of collected test cases is 9,612. We selected 100 test cases that could be executed within a day from the set using the developed method in order to answer the research questions. Table 4.1 shows the detail of the target simulator under test.

In this case study, we use two configurations of our method. Our method described in Section 4.3 uses a vector of instruction execution counts. Another version uses a 0-1 vector ignoring the number of instruction execution counts. We also use two baselines: *Random selection* and *Component-based selection*. In Random selection, we arranged the test cases in order when they were executed, and selected to become chronologically random. Component-based selection is the method used by software developers. We selected test cases so that each simulation component's number of test cases was as equal as possible. In this case study, we select 7 or 8 test cases for each of fourteen simulation components, 100 in total.

Table 4.2: Result of the Coverage.

Test Case Selection Method	Relative Instruction Coverage based on All Test Cases
Clustering-based selection (w/ count)	99.76%
Clustering-based selection (w/o count)	99.88%
Random selection (Baseline1)	65.72%
Component-based selection (Baseline2)	97.73%
All Test Cases Coverage	100.00%

4.4.1 RQ1: Does our test selection method provide better coverage than the existing method?

Table 4.2 shows the relative coverage of each method. In this research question, we define relative coverage of a selection method as the percentage of the instructions executed by 100 test cases selected by the method divided by the instructions executed by all 9,612 test cases. First, we can see that the accuracy of the Component-based method is much higher than the method based on the execution time for the baseline. This result indicates that the simulation component is highly related to the coverage. Both versions of our Clustering-based method outperform the baselines. In other words, we can select test cases with higher coverage than the Component-based method used in the company. The execution path differs depending on the parameters, even if the simulation component is the same. We can select test cases that cover different execution paths using the runtime information. In addition, the results without the runtime information were slightly better than those with the runtime information. This result may be because test cases with different instruction execution counts and similar coverage were classified into different clusters. This has increased the number of clusters showing the same coverage, making it difficult to select the test case that increases the coverage.

4.4.2 RQ2: How diverse is the execution time?

Figure 4.2 shows the distributions of the execution time excluding outliers for each method. Our method with execution count and the Random selection resulted in similar distributions to the distribution of all test cases. On the other hand, our method without considering execution count and the Component-based method resulted in significantly different distributions from all test cases. By using the information of instruction execution count, we distinguished complex executions that take a long time from those that do not and thus achieved a diverse distribution of execution times.

4.4.3 RQ3: Can our method classify an existing bug-related test case?

Based on the results of RQ1 and RQ2, we adapt our method to identify the test case that causes failures as described in Section 4.1. Although the 9,612 test cases

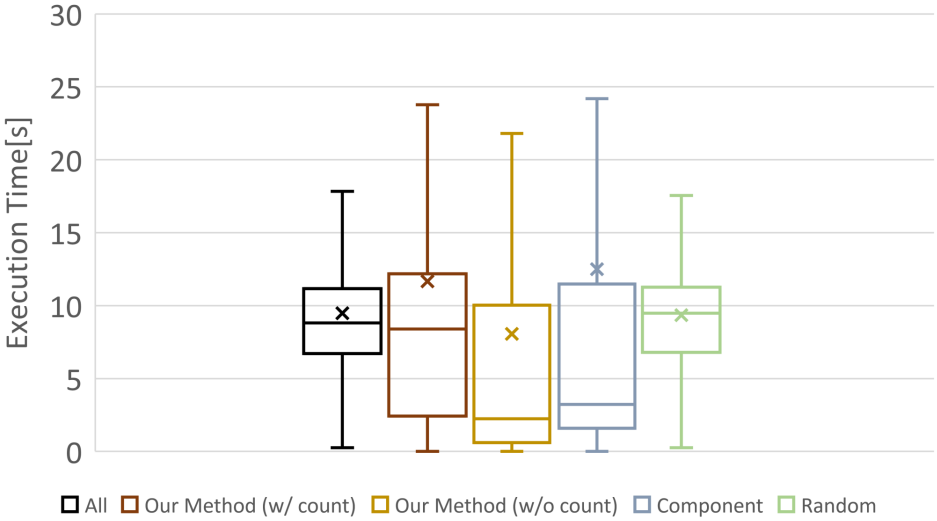

Figure 4.2: The Distribution of the Execution Time without Outlier.

we executed this case study did not contain any failures, we added one test case that caused a failure and selected one hundred test cases using our method. We adopt the method that considers the number of executions that can select test cases with high coverage and more diverse execution times that show promising results in RQ1 and RQ2.

As a result, the clustering algorithm produced a cluster including the test case that caused the failure. Consequently, the test case is successfully selected for regression testing. One of the reasons for the success of our method is that only one test case passes through the method call that causes the failure. Since our method clustering is based on the executed instructions, if an instruction is executed only in a particular test case, the test case tends to create an independent cluster. In addition, the simulation component of the test case that causes the failure is different from the other test cases, so the execution path tends to be different from the others.

4.5 Field Experiment

Based on the case study results, the software developers recognized that our method could be applied to update their dependency. They updated their JDK from AdoptOpenJDK 11 to Eclipse Temurin 17 and tested their simulator using the test cases selected by our method. We received a comment from the software developers after the update as follows:

The test execution was finished in half a day. This execution time was short enough that it did not affect the execution of the simulator. The test cases did not find any incompatibilities. The simulator has been working without failure for two weeks until now. We considered that these selected test cases contained various execution times and coverage, which are better than test cases selected by our previous Component-based method.

From this comment, we conclude that our test selection method is effective for selecting the representative test cases from a large number of test cases.

In this experiment, our method did not detect any defects. However, this does not mean that failures will not occur in the future. The most challenging part of this research is that no one can ensure that selected test cases are sufficient for regression testing. Although we evaluated our method using a known library incompatibility, the same incompatibility issue does not occur again. High code coverage also does not imply a high coverage of library usage scenarios implemented in the system. We need an advanced method to evaluate the effectiveness of test cases in the context of dependency updates.

4.6 Conclusion

We developed a test case selection method for updating the dependencies of an enterprise simulator. Our method executed test cases to create vectors based on the instructions executed in each test case and classified similar vectors. We confirmed that the test cases selected by our method achieved higher coverage and more diverse execution times than baselines. The software developers recognized the selected test cases as effective because they tested a wider range of parameters than their manually selected test cases that have been used for the simulator. The software developers also applied our method to the case of updating the JDK of the simulator and confirmed that the selected test cases sufficiently verified the compatibility.

In future work, we would like to define metrics for the diversity of execution paths and compare our method with a method that maximizes coverage, which is an extension of the conventional regression test case selection method. In addition, we would like to extend our method to automatically update test cases using new test cases added by users. We also would like to combine our method with the Component-based selection so that we can efficiently select test cases having various execution paths from tens of millions of test cases in a practical time.

Chapter 5

A Clustering-based Filtering Method for Similar Source Code Fragment Search

5.1 Introduction

In software development, developers reuse source code within a project to implement several similar functions. However, if the reused source code includes a bug, the same may occur in multiple places in the source code [55]. Therefore, when a bug is found in software, it is important to search for and fix source code fragments that are similar to the source code fragment including the bug [56].

As one of the tools to search for similar source code fragments, our research group has developed NCDSearch[1]. This tool compares source code fragments given as a query with the source code fragments cut out from the source code to be searched, and finds all the positions (file names and line numbers) of similar source code fragments. This tool searches source code fragments by giving the query in the units of line, and the shorter the number of lines given, the higher the recall. Compared to existing code clone detection tools such as CCFinderX [57], this tool misses fewer source code fragments that include similar bugs. However, this tool also finds many source code fragments that do not include bugs [31]. Ideally, the bug fix would be complete if all the detected results were examined. In actual development, it is necessary to narrow down the search results to only those code fragments that should be investigated as a priority.

In this chapter, we aim to filter the output results of NCDSearch to remove similar source code fragments that do not include bugs from the search results. We apply a code fragment clustering method proposed by Yamaguchi for filtering [58]. This method is based on the idea that when investigating a software vulnerability, a unique code fragment that is not similar to other code fragments is selected as a candidate to be investigated. Based on this idea, we extract from the search results only source code fragments that are similar to those that are likely to include similar bugs, but not similar to those that are unlikely to include similar bugs.

[1] `https://github.com/takashi-ishio/NCDSearch`

5.2 Proposed Method

The proposed method filters the set F of similar source code fragments detected by NCDSearch for query q. NCDSearch uses the normalized distance [59] for the search. The tool detects code fragments from the target source code such that the distance between each code fragment $f \in F$ and the query satisfies $d(q, f) \leq th$ for a distance threshold th.

The proposed method achieves the filtering by using neighborhood search. In concrete, we classify the source code fragments obtained from the search results into two groups: those that are strongly similar to the query and those that are not according to the threshold $d_1 (d_1 < th)$, and consider that the latter group is likely not to include any bugs. We define this set of code fragments as F' as follows.

$$F' = \{f' \in F \mid d(q, f) > d_1\} \tag{5.1}$$

We consider code fragments within distance d_2 from any of the code fragments in F' to be non-unique even if they are similar to the query, and exclude them from the search result R as follows (set $d_2 < d_1$ so that codes that match the query exactly are not excluded from R).

$$R = \{f \in F \backslash F' \mid \forall f' \in F'.\ d(f, f') > d_2\} \tag{5.2}$$

In this way, R includes only code fragments that are similar to the code fragment of the search query and different from the code fragment that is unlikely to include a bug. The final output of the proposed method is the code fragments included in R, which are output in the order of the distance $d(q, f)$ to the query.

5.3 Evaluation

We evaluate the degree of improvement in NCDSearch search results using the dataset of bug fixing cases in OSS [60]. The size of the dataset and the number of bugs are shown in Table 5.1. This dataset is a collection of cases where a single fix was made to multiple similar source codes in PostgreSQL, Git, and Linux. We perform a search on the code fragments given as a query and evaluate how many code fragments can be detected that should be modified at the same time.

The proposed method has two parameters, d_1 and d_2. We use the settings ($th = 0.50$) used in the experiment in the existing research [31], and increase d_1 and d_2 by 0.05 from 0.05 to 0.45, respectively, within the range that satisfies the relation $d_2 < d_1 < th$ described in the proposed method, and obtain the filtering result R. For the baseline of evaluation, we use the result $R' = \{f \in F \mid d(q, f) \leq d_1\}$, which is the result of filtering the NCDSearch results using only the distance threshold d_1. This setting corresponds to the case where the threshold th of NCDSearch is varied. The baseline results are obtained by varying d_1 from 0.01 to 0.50 in steps of 0.01. Assuming that the tool user checks all the execution results of all the queries, we regard the union of R obtained for each query as the filtering result of the proposed method, and the union of R' obtained for each query as the filtering result of the baseline. We use the NCDSearch version v0.3.5[2]in our evaluation.

The following indices are used for evaluation.

[2]commitID:5b76c37193741edfd41fbd7b865d04194cb3ddfa

Table 5.1: Dataset for Similar Source Code Fragments.

Project	#Query	#Bug	Median #File	Median #Lines
PostgreSQL	14	39	1,058	277,959
Git	5	8	261	67,028
Linux	34	41	22,181	6,931,715
Total	53	88	792,432	241,074,652

Table 5.2: Filtering Accuracy of the Proposed Method and Baseline.

	Precision	Recall	Reduction	Harmonic mean
Proposed Method (d_1=0.35, d_2=0.25)	0.083	0.886	0.806	0.844
Baseline ($d_1 = 0.30$)	0.081	0.773	0.827	0.799
Baseline ($d_1 = 0.31$)	0.082	0.830	0.816	0.823
Baseline ($d_1 = 0.32$)	0.077	0.841	0.802	0.821
Baseline ($d_1 = 0.35$)	0.067	0.898	0.758	0.822
Baseline ($d_1 = 0.50$)	0.018	1.000	0.000	0.000

Precision
The percentage of code fragments including bugs in the result set of the output code fragments.

Recall
The percentage of code fragments including bugs that are included in the result set.

Reduction r
The reduction ratio of source code fragments to the search result at $th = 0.50$ ($r = 1 - \frac{|R|}{|F|}$ or $1 - \frac{|R'|}{|F|}$). The higher this index is, the fewer code fragments the user has to check.

Harmonic mean of recall and reduction h
There is a trade-off between the recall and the reduction, since the more the number of output results is reduced, the higher the chance of missing a result. This harmonic mean is used to evaluate the method that significantly reduces the number of reported code fragments while maintaining a high recall.

Table 5.2 shows the best and baseline results of the proposed method based on the harmonic mean h. The total number of output code fragments for the baseline ($d_1 = 0.50$) is 4,866, in which all 88 bugs are detected (recall 1.000). In the best case, the proposed method reduces the output result by 80.6%, while 88.6% of the bugs are included in the output result. The proposed method shows a higher recall compared to the baseline ($d_1 = 0.32$) with a similar reduction rate. Compared with the baseline ($d_1 = 0.35$), which has a close value in recall, the proposed method reduces the number of output code fragments from 1,176 to 942 by about 20%. Therefore, the proposed method is more efficient than simply varying the distance threshold d_1 in the baseline method.

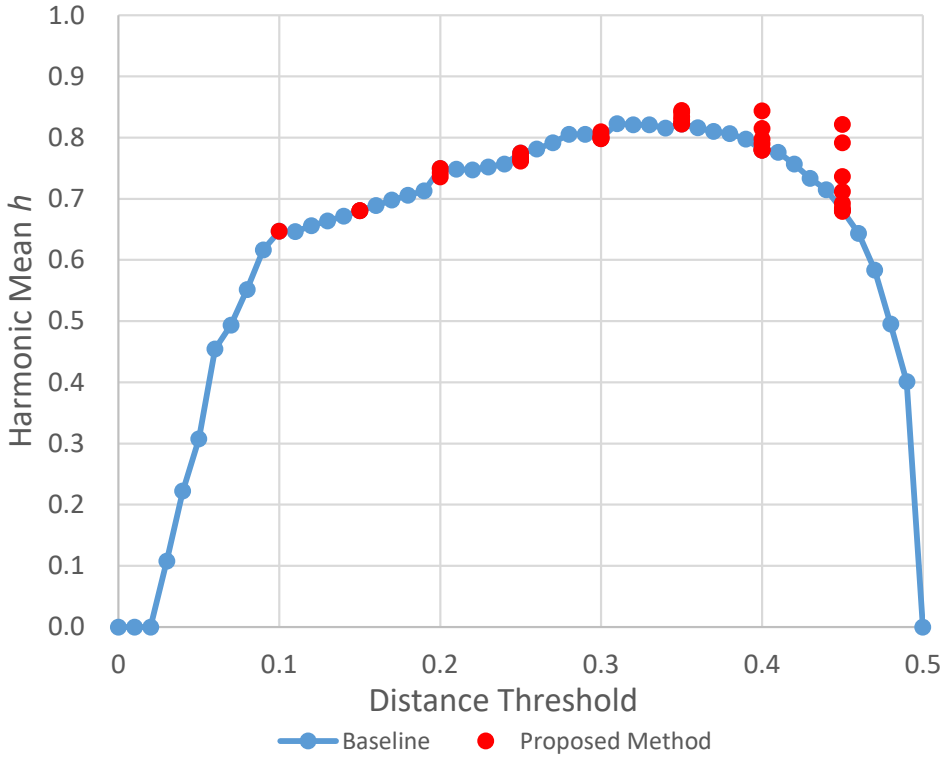

Figure 5.1: Harmonic Mean of the Proposed Method and Baseline.

The effect of parameters on the proposed method is shown in Figure 5.1. The X-coordinate of each point is d_1 for the proposed method and the baseline, and the Y-coordinate is the harmonic mean h at that time. Since several values of d_2 are used for the proposed method, all the results are plotted. From this figure, we can see that the harmonic mean h is stably the same or better than the baseline, and the proposed method is effective in improving the source code search results.

The filtering by the proposed method took less than one second per query on average. The average execution time of NCDSearch is 8.5 minutes per query, which means that the proposed method improves the output results at a low cost compared to NCDSearch.

5.4 Conclusion

In this chapter, we applied the method of vulnerability research proposed by Yamaguchi, which focused on a unique code, to a similar source code search. We defined a filtering method that extracts only the source code fragments that are different from the source code fragments that are relatively far from the query. As a result, we showed that it is possible to reduce the number of search results while maintaining a higher recall than simple filtering.

Since the dataset used in the experiment was limited to the case studies of three C projects, future work includes investigating the effects of new OSS modification

cases and the software enterprise environment [31] that uses NCDSearch.

Chapter 6

Conclusion and Future Work

6.1 Summary of Studies

This dissertation described studies on cost-effective debugging methods under restricted resources.

First, we have proposed Near-Omniscient Debugging which records execution traces with limited storage cost of Omniscient Debugging and visualizes recorded execution traces. We have conducted the quantitative evaluation of the execution trace based on the dependencies and the amount of recording and confirmed that our method requires fewer than 1% of the complete execution traces to visualize all runtime values used by 60 to 74% of instructions. We have also evaluated the completeness of recording the bug-related instruction and confirmed that Near-Omniscient Debugging could completely record the bug-related instruction for about 80 percent of bugs with a buffer size of 1024. Moreover, we applied the visualization method to actual bugs to verify its usefulness for debugging. We have shown two examples our tool can debug defects using incomplete execution traces. We have concluded that Near-Omniscient Debugging can perform Omniscient Debugging, which enables detailed dynamic analysis, using limited storage spaces. In addition, our visualization tool can be used for actual bug-fixing tasks, indicating the first step toward the practical application of Near-Omniscient Debugging.

Second, we have proposed a test selection method that uses runtime information to update the dependency. We have shown the usefulness of our test selection method in terms of higher coverage and more diverse execution times than baseline methods. We have also conducted a field experiment by applying the method to actual failure and confirmed that the selected tests sufficiently verified the compatibility. In terms of efficient identification of the cause of the failure and efficient reproduction of the failure, the proposed method can be used effectively.

Finally, we have proposed a filtering method for similar source code fragment search, which focuses on unique code. We have shown that it is possible to reduce the number of search results while maintaining a higher recall rate than simple filtering. By reducing the number of search results that the developer has to inspect, the proposed method contributes significantly to reducing the debugging effort.

6.2 Future Work

We have confirmed that Near-Omniscient Debugging can completely record bug-related instructions for the most bugs. We should specify the appropriate buffer size depending on the domain and software characteristics for practical usage of the execution trace, so we would like to consider the method to determine the parameter to support it. As a method of using the recorded traces, we have tackled with debugging support through visualization in this work. In the future work, as with Omniscient Debugging, we will consider applying Near-Omniscient Debugging to more detailed analysis, such as visualizing dependencies and reproducing partial executions.

We have proposed a test selection method that uses runtime information to update the dependency. When test cases are added or software is updated, the selected test cases needs to be updated. It is required to realize an automatic test case update method that supports various execution for future work. It is also a future work to determine the appropriate number of test cases, taking into account the execution time of each test case.

We have proposed a filtering method for similar source code fragments. The proposed method is applied to a limited number of languages and similarity indices. Therefore, it is necessary to consider a similarity index suitable for each language and an appropriate parameter determination method according to the reduction rate allowed by the developers.

Bibliography

[1] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible Debugging Software. Judge Business School, University of Cambridge, Cambridge, U.K., Technical Report, 2013.

[2] Evans Data Corporation. Worldwide professional developer population of 24 million projected to grow amid shifting geographical concentrations. Evans Data Corporation, 2019.

[3] Ian Sommerville. *Software Engineering 5th Edition*. U.S:Addison-Wesley, 1995.

[4] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., USA, 1980.

[5] Undo Software. Increasing software development productivity with reversible debugging. Undo Software, Technical Report, White Paper, 2014.

[6] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 60–70, 2018.

[7] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[8] Ganesh J. Pai and Joanne Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on Software Engineering*, 33(10):675–686, 2007.

[9] John C. Munson and Taghi M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.

[10] Craig S. Wright and Tanveer A. Zia. A quantitative analysis into the economics of correcting software bugs. In *Proceedings of the 4th International Conference on Computational Intelligence in Security for Information Systems*, pages 198–205, 2011.

[11] NIST Rep. The economic impacts of inadequate infrastructure for software testing. `http://www.abeacha.com/NIST_press_release_bugs_cost.html`, 2002.

[12] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM SYSTEMS JOURNAL*, 41:4–12, 2001.

[13] Andreas Zeller. *Why programs fail, the 2nd edition.* O' Reilly Japan, 2012.

[14] Monika A. F. Müllerburg. The role of debugging within software engineering environments. In *Proceedings of the Symposium on High-Level Debugging*, page 81–90, 1983.

[15] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[16] Boyuan Chen and Zhen Ming (Jack) Jiang. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*, pages 71–81, 2017.

[17] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, 2014.

[18] Apache log4j 2. `https://logging.apache.org/log4j/2.x/`.

[19] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* John Wiley & Sons, Inc., USA, 1996.

[20] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, pages 572–583, 2018.

[21] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[22] Bil Lewis. Debugging backwards in time, CoRR, cs.SE/0310016, 2003.

[23] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[24] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, pages 361–372, 2014.

[25] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[26] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[27] Chen Zhang, Zhenyu Chen, Zhihong Zhao, Shali Yan, Jinyu Zhang, and Baowen Xu. An improved regression test selection technique by clustering execution profiles. In *Proceedings of the International Conference on Quality Software*, pages 171–179, 2010.

[28] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 535–552, 2007.

[29] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, pages 91–100, 2019.

[30] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. Fastlane: Test minimization for rapidly deployed large-scale online services. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 408–418, 2019.

[31] Takashi Ishio, Naoto Maeda, Kensuke Shibuya, and Katsuro Inoue. Cloned buggy code detection in practice using normalized compression distance. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, pages 591–594, 2018.

[32] Diomidis Spinellis. *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional, 2016.

[33] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[34] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, pages 347–362, 2011.

[35] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 154–164, 2016.

[36] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *SIGARCH Comput. Archit. News*, 39(1):3–14, 2011.

[37] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33, 2014.

[38] Bas Cornelissen, Leon Moonen, and Andy Zaidman. An assessmentmethodology for trace reduction techniques. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 107–116, 2008.

[39] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *Proceedings of the 26th International Conference on Software Engineering*, pages 512–521, 2004.

[40] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[41] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 911–922, 2016.

[42] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, 2004.

[43] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 253–257, 2014.

[44] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[45] Giuseppe Cattaneo, Pompeo Faruolo, Umberto Ferraro Petrillo, and Giuseppe F. Italiano. JIVE: Java interactive software visualization environment. In *Proceedings of the IEEE Symposium on Visual Languages - Human Centric Computing*, pages 41–43, 2004.

[46] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, pages 211–224, December 2003.

[47] Nima Honarmand and Josep Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*, pages 455–456, 2014.

[48] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, page 377–389, 2017.

[49] Salman Mirghasemi, John J. Barton, and Claude Petitpierre. Querypoint: Moving backwards on wrong values in the buggy execution. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, page 436–439, 2011.

[50] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, March 2004.

[51] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 112–124, 2020.

[52] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: A study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 215–225, 2017.

[53] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 112–122, 2018.

[54] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[55] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 73–88, 2001.

[56] Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie. Transferring code-clone detection and analysis to practice. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 53–62, 2017.

[57] Toshihiro Kamiya. AIST CCFinderX. `http://www.ccfinder.net/ccfinderxos.html`.

[58] Fabian Yamaguchi. Pattern-based vulnerability discovery. Ph.D. Thesis, University of Gottingen, 2015.

[59] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M.B. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004.

[60] Jingyue Li and Michael D. Ernst. CBCD: Cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering*, page 310–320, 2012.