Near-Omniscient Debugging for Java Using Size-Limited Execution Trace

Kazumasa Shimari*, Takashi Ishio†, Tetsuya Kanda*, Katsuro Inoue*
* Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
Email: {k-simari, t-kanda, inoue}@ist.osaka-u.ac.jp
† Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
Email: ishio@is.naist.jp

***Abstract*—Logging is an important feature for a software system to record its run-time information. Detailed logging allows developers to collect information in situations where they cannot use an interactive debugger, such as continuous integration and web application server cases. However, extensive logging leads to larger execution traces because few instructions could be repeated many times. To record 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. Our tool monitors a Java program's execution and annotates source code with observed values in an HTML format. Developers can easily investigate the execution and share the report on a web server. In case of DaCapo benchmark applications, our tool requires fewer than 1% of the complete execution traces to visualize all runtime values used by 66% of instructions that are executed less than 64 times. Developers also can obtain data dependencies with precision 91.8% and recall 79.0% using this tool.**

***Index Terms*—Dynamic Analysis, Logging, Software Visualization**

I. Introduction

Debugging is a methodology to identify defects in source code from the diagnosis of its external behavior. For efficient debugging, developers monitor the execution order of instructions and actual values of variables in source code [12]. Additionally, developers may compare the program behavior at various points of executions where a failure does or does not occur [2], [6]. Interactive visualization tools such as JIVE [4] and break-point debuggers are useful for such analysis; however, developers struggle to use those tools for systems running on continuous integration and web application servers.

Logging is a common practice to record a program execution as a sequence of messages reporting the software's progress and its important data [7]. However, logging recorded in production may not contain sufficient data for debugging since the data is specified at development time [14]. To enable efficient debugging, an automatic method capable of recording a program's execution in detail is needed.

Omniscient debugging [8] is a technique used to record all the runtime events during a program execution. Although the technique allows developers to inspect the state of a program at an arbitrary point in execution, it results in a huge execution trace, in some cases growing as fast as 10 MB per second [11]. 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 [3].

In this work, we propose Near-Omniscient Debugging to record and visualize an execution trace within limited storage space constraints. Since a full execution trace includes many uninteresting method calls such as utility functions [3], 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.

Our implementation records local variables and fields used in a Java program's execution and annotates source code with the recorded values. Our tool also provides a filtering feature to display the values recorded in specific intervals. This filtering feature allows developers to investigate both control-flow and data-flow among instructions.

In the remainder of the paper, Section II explains the background of the tool and Section III describes its implementation. Section IV shows the performance of the tool. Then, Section V describes a use case example. Finally, Section VI concludes the paper and describes future work.

II. Background

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. [13] proposed an effective compression method tailored for execution traces comprising a sequence of memory address accessed by a program. This method employs delta encoding because programs often repeat the same instruction 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. [3] 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.

2576-3148/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSME.2019.00068

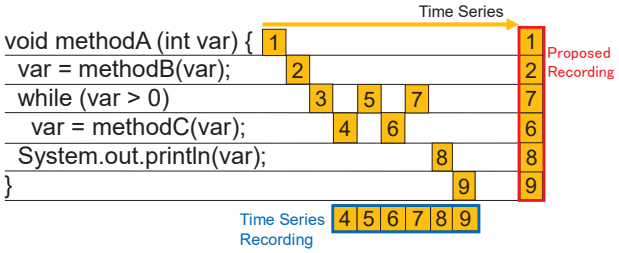


Fig. 1. The proposed recording image

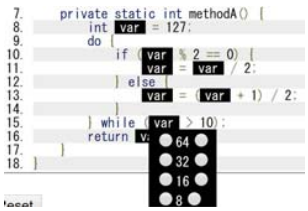


Fig. 2. Trace view

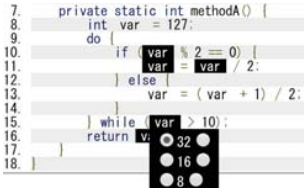


Fig. 3. Trace view with filtering

Hizrel et al. [5] 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.

Another approach to minimizing runtime overhead and storage space is a specialized execution trace tailored for a specific purpose. Liu et al. [9] 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. [15] 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.

III. Proposed Method

Our tool records a partial execution trace of a Java program and generates HTML files to interactively explore the recorded trace. Figure 1 illustrates the key idea of our tool. In this figure, we have a limited storage space that can record up to six steps of program execution. Suppose an execution comprises of nine steps, as indicated by numbers in yellow boxes. A naive time-series logging records the last six steps as indicated in the bottom of the figure. On the other hand, our tool prepares buffers for each line of code in order 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.

Our tool comprises two components: Near-Omniscient Trace Recorder and Interactive View Generator. The following subsections explain each component in detail.

A. Near-Omniscient Trace Recorder

Our recorder component is an extension of the existing trace recorder for REMViewer [10], 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.

The tool injects logging instructions to a target program using bytecode instrumentation. An execution trace is a sequence of events $\langle d, t, v \rangle$ where d represents the data element (e.g., a local variable) used by an instruction, t represents the thread that executed the instruction, and v represents the observed value. It is worth noting that an instruction may have several data elements, e.g., arguments of a method.

Our tool records the latest k events for each data element d. Then it takes buffer size k as a parameter, allocates buffers for each data element, and accumulates the observed events alongside their timestamps. 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 method with $k = \infty$ is conceptually equivalent to omniscient debugging.

To enable users to investigate how objects are manipulated, our tool assigns object IDs for each object reference. 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 is considered as future work.

B. Interactive View Generator

The interactive view generator translates an execution trace obtained by the trace recorder into an interactive view represented by HTML files corresponding to each source. This view displays the source code, whose variables are highlighted. The variables are linked to the actual values recorded in the trace. By hovering a mouse cursor on a highlighted variable, the values of the variable are displayed. For example, Figure 2 shows a screenshot of an interactive view displaying the values of `var` in ascending order by time.

This interactive view can filter values by specifying a time interval. For example, in Figure 2, a user can click on radio buttons shown in the left and right sides of each value. A click on the left button specifies a start point of an interval while a click on the right one specifies an end point. Figure 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

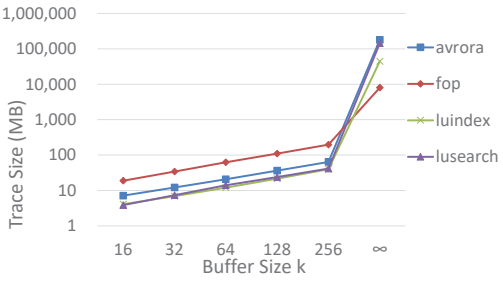


Fig. 4. Data size of execution traces

TABLE I
SIZE OF BENCHMARK PROGRAMS AND THEIR EXECUTION TRACES

Name	#Class	#Method	#Element	Size(k=∞)	Size(k=64)
avrora	527	3,456	122,513	174.2 GB	7.2 MB
fop	1,161	10,178	433,810	7.8 GB	18.8 MB
luindex	231	2,467	111,643	43.5 GB	4.1 MB
lusearch	199	2,140	88,785	140.3 GB	3.9 MB

for that variable is turned off. In the figure, the highlighting for `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.

IV. PERFORMANCE

We evaluated the performance of our tool under the following aspects: size of the execution traces, time required to collect them, and their accuracy. To evaluate those aspects, we used full execution traces generated by omniscient debugging ($k = \infty$) as a baseline. We selected four benchmarks that could complete executions of DaCapo Benchmarks 9.12-bach [1]. Then, we executed our tool in order to collect execution traces with varying values of k: 16, 32, 64, 128, 256, and ∞. The obtaining traces contain standard library calls in the target benchmarks but do not contain the internal behavior of the Java standard libraries. Performance was measured on Windows 10 running a Xeon(R) W-2123 @3.60 GHz processor, 32 GB DRAM, and an HDD.

A. Size of Execution Traces

Figure 4 shows the storage size of execution traces while varying k. Table I shows the program size of each benchmark. The column #Class is the number of loaded classes that executed the benchmark, excluding the Java standard libraries identified by the prefix of the package names (`java`, `javax`, and `sun`). The columns #Method and #Element show the numbers of methods and data elements contained in classes instructions, including both executed and non-executed components. The columns "Size ($k = \infty$)" and "Size ($k = 64$)" indicate the size of execution traces respectively with a k-value of ∞ and 64. The traces for our tool consume less than 1% of the space taken by full traces.

TABLE II
EXECUTION TIME OF OUR TOOL (MILLISECONDS)

Name	Normal	Near-Omni. (k=64)	Omni.
avrora	9,397	2,229,917	6,704,254
fop	1,528	32,549	129,229
luindex	6,049	89,556	608,175
lusearch	7,496	1,870,915	5,508,298

TABLE III
EVALUATION OF DATA DEPENDENCIES (SUM OF FOUR BENCHMARKS)

k	#Depend.	Precision	Recall	F-measure	Complete ratio
16	41,184	0.918	0.748	0.824	0.565
32	42,130	0.920	0.767	0.837	0.596
64	43,476	0.918	0.790	0.849	0.662
128	44,425	0.917	0.807	0.858	0.696
256	45,528	0.916	0.826	0.869	0.725
∞	50,502				

B. Time required to Collect Execution Traces

Table II shows the execution time of benchmarks with our trace recorder. The columns show the execution time without recording, with our recording ($k = 64$), and with recording for omniscient debugging ($k = \infty$), respectively. Our Near-Omniscient Debugging takes 130 times longer than normal executions on average but is faster than omniscient debugging because smaller traces require a shorter time to be recorded in storage.

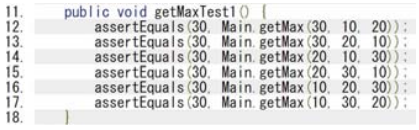
C. Accuracy of Data Dependencies in Execution Traces

To analyze the accuracy of our trace, we evaluated data dependencies among instructions. The data dependency is defined as an assignment-reference instruction pair that accesses the same memory address using local, field, or array variables. Table III shows the number of data dependencies varying k and its precision, recall, and F-measure. The table shows that our method achieves high precision and recall with averages of 0.9 and 0.8 respectively, using fewer than 1% of the space consumed by full execution traces. The right-most column, "Complete ratio", indicates the ratio of instructions that are executed less than k times; i.e. values for those instructions are completely recorded. In case of $k = 64$, the trace retains variable values for 66% of all instructions in the programs.

V. USE CASE EXAMPLE

To demonstrate the usefulness of the tool, we perform the debug of a small program. The program's goal in this case study is to select the maximum number from three given numbers. Figure 5 shows test cases of the target method that provide parameters 10, 20, and 30 in different orders to the target method. 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 target method, we collect an execution trace of the test and generate interactive views of the program using the following commands.



```
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.     }
```

Fig. 5. Test cases of a program

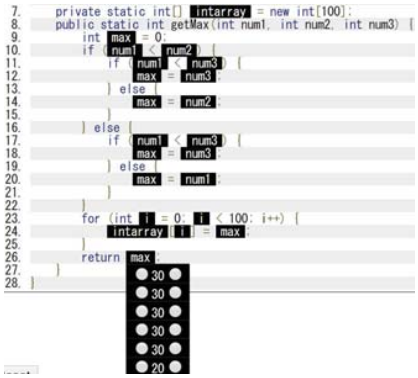


Fig. 6. The interactive view of use case

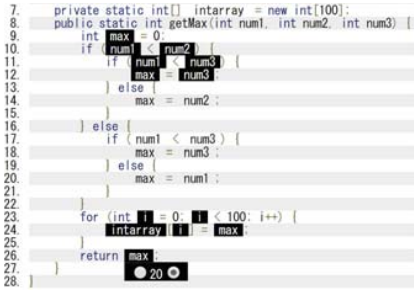


Fig. 7. The result of the filtering

```
java -javaagent:NearOmniTracer.jar=size=32,
  output=trace-dir -jar getMax.jar
java -jar NearOmniVis.jar trace-dir *.java
```

The first command traces a program with our recorder. The second command translates the generated trace and source files into HTML files.[1]

Figure 6 shows the view of the target method. We can see that all cases return a value on the variable `max` at line 26. 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 `max` at line 9 to the final value of `max` at line 26, by clicking on the left radio button of the sixth value at line 9 and the right button at line 26. Figure 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.

VI. Conclusion

We proposed Near-Omniscient Debugging to monitor and visualize detailed software with reducing storage space consumption. Our method takes as input k to specify the number of recorded values for each instruction. It retains actual values for 66% of instructions and accurate data dependencies using fewer than 1% of the full execution traces. Developers can use our tool to monitor remote program execution, such as testing on continuous integration servers, and to visualize the behavior of test failures.

As future work, we would like to investigate effective logging for textual contents such as strings and exceptions. We also would like to evaluate the usability of the tool and confirm that whether the accuracy of data dependencies is not 100% affects the reliability of the debugging process.

[1]The generated view files and tools are available at http://sel.ist.osaka-u.ac.jp/people/k-simari/ICSME2019/

Acknowledgments

We thank anonymous reviewers, Raula Gaikovina Kula, and Davide Pizzolotto for their editorial help.

This work has been supported by JSPS KAKENHI Nos. JP18H03221, JP18KT0013, JP18H04094 and JP19K20239.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. OOPSLA*, 2006, pp. 169–190.

[2] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. ICSE*, 2005, pp. 342–351.

[3] B. Cornelissen, L. Moonen, and A. Zaidman, "An assessment methodology for trace reduction techniques," in *Proc. ICSM*, 2008, pp. 107–116.

[4] P. V. Gestwicki and B. Jayaraman, "JIVE: Java interactive visualization environment," in *Companion Proc. OOPSLA*, 2004, pp. 226–228.

[5] M. Hirzel and T. Chilimbi, "Bursty tracing: A framework for low-overhead temporal profiling," in *Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[6] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential Slicing: Identifying Causal Execution Differences for Security Applications," in *Proc. of the 32nd IEEE Symposium on Security and Privacy*, 2011, pp. 347–362.

[7] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *Proc. MSR*, 2016, pp. 154–164.

[8] B. Lewis, "Debugging backwards in time," CoRR, cs.SE/0310016, 2003.

[9] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *Proc. ICSE*, 2016, pp. 911–922.

[10] T. Matsumura, T. Ishio, Y. Kashima, and K. Inoue, "Repeatedly-executed-method viewer for efficient visualization of execution paths and states in java," in *Proc. ICPC*, 2014, pp. 253–257.

[11] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proc. OOPSLA*, 2007, pp. 535–552.

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

[13] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing java programs," in *Proc. ICSE*, 2004, pp. 512–521.

[14] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 3–14, 2011.

[15] X. Zhang, R. Gupta, and Y. Zhang, "Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams," in *Proc. ICSE*, 2004, pp. 502–511.