VeDebug: Regression Debugging Tool for Java

Ben Buhse[1], Thomas Wei[1], Zhiqiang Zang[1], Aleksandar Milicevic[2], and Milos Gligoric[1]
[1]The University of Texas at Austin, [2]Microsoft
bwbuhse@utexas.edu, thomas.wei@utexas.edu, zhiqiang.zang@utexas.edu, almili@microsoft.com, gligoric@utexas.edu

***Abstract*—Developers spend substantial time debugging their programs, yet debugging is still one of the most tedious activities. Interactive debuggers have been around for as long as computing, but the way they are used—set a breakpoint, reason about the state, step into/over—has not substantially changed. The last big discoveries, which happened decades ago, include visual debugging (e.g., DDD) and time-travel debugging. Although existing interactive debugging tools provide useful and powerful features, they are limited to a single program execution, e.g., a developer can only see data values and navigate the control flow of a single program execution at a time.**

We present VEDEBUG, the first video-based time-travel regression debugging tool to advance users' debugging experience. VEDEBUG introduces two unique features: (1) regression debugging, i.e., setting a "divergence breakpoint" (which "breaks" the execution whenever the control flow of the current execution diverges from the flow of a previously captured execution), and (2) video debugging, which provides features similar to those of a video player (e.g., speed up/slow down the replay). The demo video for VEDEBUG can be found at: https://www.youtube.com/watch?v=lOiGrE_sc10.

I. INTRODUCTION

Debugging is a tedious and time consuming activity [12]. Various companies have reported that debugging can take more than 50% of development time [8]. Researchers and practitioners have developed various tools to cope with challenging debugging tasks.

Traditional *interactive debuggers*, such as GDB and JDB, provide numerous features, including setting a breakpoint, defining watches, exploring stack frames, dumping memory, etc. More exotic debugging tools may provide support for interrogative debugging (e.g., Whyline [7], which can answer "why did" and "why didn't" questions about the program state), or interactive graphical data display (e.g., DDD [3]).

Although existing debugging tools provide useful and powerful features, they are limited to a *single program execution*, e.g., a developer can only see data values and navigate the control flow of a single program execution. This approach does not closely match a common debugging workflow when a developer is trying to understand why a program that "worked" in the *past* doesn't "work" *now*. To successfully complete this task, the developer must understand the impact of code changes on their data and control flow [12].

To enable more effective debugging and help developers understand the impact of their code changes, we present VEDEBUG—the first tool for *interactive regression debugging*. VEDEBUG maintains a database of historical execution traces and provides features for comparing (any two) traces and data values from those traces. Specifically, VEDEBUG enables a user to step through *divergence* and *reconvergence* points. Additionally, VEDEBUG is the first to offer an interface that resembles video players with features such as playing the execution forward and backward, adjusting the speed, pausing the execution, etc., while still being able to inspect the stack frame and execution trace at any point during the video.

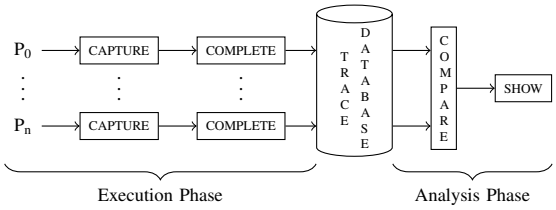

Fig. 1: Overview of VEDEBUG workflow

We implemented VEDEBUG for Java by dynamically instrumenting code to capture execution traces and method argument values. This paper describes the details of our prototype and the design of planned studies. Our prototype is available at https://github.com/EngineeringSoftware/VeDebug.

II. VEDEBUG TOOL

VEDEBUG combines elements of *interactive* and *time-travel* debugging with a video-like interface. As with most modern profilers, VEDEBUG captures a trace of the entire program execution before processing the trace for playback and analysis. This *post mortem* technique for debugging was what allowed us to create the video-like interface, as well as what created the opportunity for regression debugging. By comparing the traces of two different executions and locating divergences and reconvergences in their behavior, VEDEBUG provides the user with unique insight into the effects of changes between the two versions and how these effects propagate throughout the execution; the traces that are compared can be obtained for any two versions, which are potentially many commits apart.

Figure 1 shows the workflow of VEDEBUG, which consists of two phases: execution and analysis. The *execution phase* takes as input a program under debugging and a command to execute (e.g., run a test case). During the execution of the command, VEDEBUG captures an execution trace. To optimize the trace capturing, VEDEBUG captures a short trace, which can be used to infer all executed statements; we run an extra step to complete the trace. The completed trace is stored in a *trace database*. A user can go through the execution phase any number of times before deciding to compare and visualize traces and values. The *analysis phase* takes as the input any

two traces ("old" and "new") from the trace database, which are selected by a user, compares the two traces, and visualizes the new trace while enabling regression debugging features with respect to the old trace.

A. Trace Capturing

VEDEBUG traces contain all of the necessary information to completely reconstruct a program's execution from start to finish. This information consists of when methods are called and when they are returned from, as well as when basic blocks are entered and, by extension, when they are exited. By capturing these points, we can reconstruct a sequence of executed statements.

We used `java.lang.instrument` to intercept the loaded but unmodified byte arrays of each class file. Next, we altered the bytecode belonging to each class by using the ASM framework for bytecode manipulation. This allowed us to insert statements at strategic points in the bytecode for the *dynamic* collection of the necessary data; during this process we also do *preprocessing* of bytecode to save metadata for each class that will help us to obtain the complete trace.

Recording traces for every single class loaded into the JVM would be suboptimal in several respects: a large amount of irrelevant information would be captured (such as traces from the JDK classes) and the overhead would be more significant. Instead, VEDEBUG utilizes a whitelist/blacklist system: certain packages are ignored by default, but the user has the ability to refine both lists and precisely specify which packages should be instrumented.

During the bytecode transformation, VEDEBUG uses ASM's Visitor API to perform both preprocessing and dynamic analysis. The preprocessing includes analysis of method signatures, beginning and end line numbers of methods, line numbers of *potential* method invocation sites, and the line numbers of basic blocks. At the same time, there are instructions being inserted at strategic points in the bytecode to be used for dynamic collection of data. During the dynamic analysis, VEDEBUG gathers values of method arguments and return values, line numbers of *actual* method invocation sites (using the aforementioned potential invocation sites), and locations where the execution changes basic blocks. In order to minimize overhead, string and int literals are saved with the ASM library during static analysis and are collected when that portion of the code is executed. This means that VEDEBUG simply needs to load the specific literal rather than recalculating various line numbers and method names every time a method is called. Meanwhile, the method signature is parsed and each argument is loaded and then sent to the correct method for saving that type. For return values, the top value on the JVM stack (or top two values in the cases of longs and doubles) is duplicated and the correct saving method is called just like for arguments.

VEDEBUG is able to avoid any *major* slowdowns which would cause unreasonable overhead by giving each method a unique ID during transformation. When each method is called or returned from, that ID along with any arguments the method may have, are the only things that are stored, as opposed to the entire method signature every time. The IDs are all kept in a separate file which is later parsed by an intermediate script during trace completion to replace each ID with its proper signature, including the method's line numbers.

B. Trace File Formats

As mentioned above, VEDEBUG creates a number of different files while capturing the execution trace of a program. These are MethodIDs, MethodCalls, and individual files named classNameBB that contain the list of line numbers of basic blocks for all methods in a single class.

1) MethodIDs: For MethodIDs, only a numbered list of every instrumented method in the program is stored; one line per method. Each line is formatted as a unique ID, followed by the starting and ending line numbers of the method, then the source file which the method is in, and then the class name, and method name. Finally, at the end of each line are the methods parameters and lastly its return type. Below is an example line from MethodIDs obtained while running `ValueGraphTest` from Google Guava [4]:

2 37 51 com/google/common/graph/ValueGraphTest.java com/· · ·/ValueGraphTest edgeValue_missing − V

The MethodIDs file is used to (more) efficiently store each method call in the MethodCalls file by only saving a method ID that can be referenced later rather than saving the entirety of every method's information any time a method is called.

2) MethodCalls: The MethodCalls file contains information about the calls and returns from every instrumented method as well as information about every time a new basic block is entered within a method. Lines that are reporting a call to a method start with the invocation line number of a method (or -1 if the method was called implicitly at the start of execution, such as main or static initializer blocks), along with the method's unique ID from MethodIDs. If the method has parameters, the line will also include the values for those parameters. For method returns, the line starts with a dash followed by the method ID and, for non-void methods, a return value. Lastly, the line numbers for indicating basic block changes are started with an @ followed by the ID of the method which the basic block is in, and then a colon and the line number where the basic block starts. Below are several lines from the MethodCalls file obtained while running `ValueGraphTest` from Guava:

39 46 "default" \n @46:132 \n @46:131 \n − 46 "default"

3) classNameBBs files: The last format of file created by the Java side of the program is the classNameBBs format where basic blocks are saved. This format is the simplest, only consisting of a list of every line number where a basic block starts in each class. The ending line numbers of basic blocks are not needed because every basic block ends either when a new basic block starts or at the end of the method. This is why the MethodCalls file only contains the line numbers of methods which are being entered instead of the line numbers for both the entrance and exit of basic blocks.

Fig. 2: VEDEBUG debugging environment: current statement and context (top-left), argument values and video status (bottom-left), current stack trace (top-right), and execution trace and the current position in the trace (bottom-right)

C. Trace Completion

Once the trace has been captured, we use an intermediate script to integrate all the information acquired by the instrumentation into a single processed trace. This completed trace is what is used for trace comparison and eventually loaded by the visualizer. During the process of trace completion, all of the method call and change of basic block information collected during the execution is compiled into line number intervals and call stack operations that can be mapped to the proper source code file by VEDEBUG during visualization.

To track the propagation of exceptions in post-analysis, we use the basic block transition information gathered in the trace. Whenever an inconsistency in the transitions is found, we pop frames of the call stack until the inconsistency is resolved or the execution ends because of the exception. This is done in trace completion rather than trace capturing because Java exceptions use many different bytecode instructions, making it extremely difficult to design a catch-all system for logging exception throws and returns.

D. Trace Comparison

To facilitate regression debugging, VEDEBUG has the capability to take two traces from any point in the evolution of a program and extract key differences in the behavior of the given versions. The first trace given is the faulty trace or, in general, the trace under inspection and what will be referred to as the main trace. The other trace is the reference trace, the trace for the main trace's behavior to be compared against. The trace comparison process yields a modified version of the main trace with annotations of the points of *divergence* and *reconvergence* with the behavior of the reference trace and, optionally, changes between method arguments and return values during the intervals where the two executions are in sync as well.

Trace comparison algorithm is designed as follows. Starting from the beginning of both traces, move in lockstep through the traces as long as they are in sync. The first encountered difference constitutes a divergence point. From that point on, the two traces are considered out of sync. A list of possible indices within the reference trace where the two traces can reconverge should be obtained by advancing through the reference trace until the point where the function in which the initial divergence took place is removed from the stack. Then step through the main trace, checking if the trace element is a match to any of the potential match candidates. A match constitutes a reconvergence point, meaning that the two traces can be resynced with an appropriate offset and the process can continue with finding the next point of divergence.

E. User Interface

Figure 2 shows a screenshot of the VEDEBUG GUI while running `ValueGraphTest` from Guava. Note that the entire interaction with the debugger is via a keyboard; several keybindings are available (with Emacs-like being the default).

As VEDEBUG displays the execution of the program as a video, controls analogous to those of other video players were developed. These include options to vary the rate of playback, play the program's execution in reverse, and skip to significant points in the execution. Since VEDEBUG retains information about the state of the stack and control flow in the completed trace used to play back the execution, standard interactive debugger features such as step over, step out, and

TABLE I: Tracing Overhead Introduced by VEDEBUG

Project	Test Case	Execution [ms]	
		Default	VEDEBUG
Lang	NoClassNameToStringStyleTest	14	168
	ConstructorUtilsTest	18	155
	ConstantInitializerTest	4	27
	ShortPrefixToStringStyleTest	12	176
	SimpleToStringStyleTest	12	164
JFreeChart	LayeredBarRendererTest	1578	1601
	CategoryTickTest	102	108
	DateTickTest	30	90
	MonthDateFormatTest	161	200
	CategoryTextAnnotationTest	120	169
la4j	BasicVectorTest	81	642
	GaussJordanInverterTest	5324	5431
	SeidelSolverTest	68	165
	GaussianSolverTest	55	131
	JacobiSolverTest	59	161

alike are still available to the user. The user is also able to search for the next time the control flow reaches a specific function. Due to VEDEBUG's time-travel nature, all of these functions are available in the reverse direction as well. In addition to the visual trace through the source code, the user interface also displays information relevant to the program's execution. The stack trace is shown on the right-most window and the parameters to and return value of the current function are displayed in the window below that of the main video. These give the user information about the immediate state of the program as most interactive debuggers do. However, unlike traditional debuggers, VEDEBUG also provides the user with a sequential list of past and future events of interest that give context to the immediate state of the program within the execution as a whole. The list of events of interest include function calls, returns, exceptions, and divergence and reconvergence points (for the case of regression debugging).

F. Implementation

We implemented VEDEBUG in Java 8, Python, and C. We use Java to manipulate the Java bytecode, capture dynamic trace data, and store the trace to disk. We use Python to perform trace completion and trace diffing. Finally, we use C and the ncurses library [10] to implement the GUI. Basically, the entire GUI runs in a terminal and is independent on any existing IDE. Moreover, our design, which sets a clear boundary between tracing and visualization, will enable easy development of interactive regression debuggers for any other programming language (e.g., C or Scala), as long as traces are stored in the format defined by VEDEBUG.

III. CASE STUDIES

We performed an initial evaluation of VEDEBUG by running several tests from three open-source projects. Our goal was to measure the overhead of capturing execution traces. Table I shows the names of the projects and tests (which we randomly selected), execution time of the tests without using VEDEBUG, and execution time with trace capturing; we did not capture arguments with non-primitive types. We can observe that VEDEBUG introduces non-negligible overhead, but we believe that this cost is acceptable during debugging [2]. In our future work we plan to further optimize trace capturing and perform a user study to evaluate our GUI design and regression debugging features. Our plan for the study is to ask users to debug several bugs available in an existing bug databases, e.g., Defect4J; we plan to report similar metrics as prior work on interactive debugging [7].

IV. RELATED WORK

There has been a large amount of work on understanding user interactions with IDEs (e.g., [6]), interactive debugging (e.g., [1], [3], [7]), and automated debugging (e.g., [9], [11], [12]). For example, Afzal and Le Goues [1] studied impact of debugging features on debugging time and effort; Gu et al. [5] introduced a DSL for searching for prior bugs based on execution traces; Ko and Myers [7] introduced interrogative debugging. We are the first to propose interactive regression debugging and implement a prototype tool with this feature.

V. CONCLUSION AND FUTURE WORK

We presented VEDEBUG, a video-based time-travel regression debugging tool to advance users' debugging experience. VEDEBUG brings two unique features: (1) regression debugging, i.e., setting a breakpoint based on prior run so that the debugger automatically breaks as soon as a divergence from a previous trace is detected, and (2) video debugging, which provides functions similar to a video player.

In the future we plan to visualize two traces side-by-side, visualize object graphs in a better way than using a flat list of values, and capture values of local variables.

Acknowledgments. We thank Ahmet Celik, Pengyu Nie, Karl Palmskog, Marinela Parovic, and Chenguang Zhu for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1566363 and CCF-1652517.

REFERENCES

[1] Afsoon Afzal and Claire Le Goues. A study on the use of IDE features for debugging. In *MSR*, pages 114–117, 2018.
[2] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, pages 154–163, 2006.
[3] DDD - DataDisplay Debugger. https://www.gnu.org/software/ddd.
[4] Google core libraries for Java. https://github.com/google/guava.
[5] Zhongxian Gu, Earl T. Barr, Drew Schleck, and Zhendong Su. Reusing debugging knowledge via trace-based bug search. pages 927–942, 2012.
[6] Zhongxian Gu, Drew Schleck, Earl T. Barr, and Zhendong Su. Capturing and exploiting IDE interactions. In *Onward!*, pages 83–94, 2014.
[7] Andrew Jensen Ko and Brad A. Myers. Designing the Whyline: a debugging interface for asking questions about program behavior. In *CHI*, pages 151–158, 2004.
[8] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *ESEM*, pages 383–392, 2013.
[9] Kivanc Muslu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *FSE*, pages 631–634, 2013.
[10] ncurses. https://www.gnu.org/software/ncurses.
[11] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.
[12] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, pages 253–267, 1999.