Thuy Nhi Tran-Thien, Hang Bui, Eric Guo, Boya Cao

FootPrint

Motivation

The standard IDE debuggers such as those in IntelliJ and Eclipse do not offer the option of stepping backward through the execution. Occasionally, programmers set the breakpoint too far or step through the debugger too fast and miss the step they wanted to examine and have to restart. Furthermore, having to keep trying to reproduce a bug can be frustrating as well. Having the option to view your variable's history would help solve these issues. This is what we are trying to achieve with FootPrint: a user-friendly, lightweight, and simple way for Java developers to view the history of their variables.

For example, imagine a CS student who is trying to debug their program. They want to look at what is in their array at line 15, but they accidentally advance to line 16 when the array is discarded. With FootPrint, they could simply examine the state of the selected array by looking at the program state from line 15, which FootPrint would store for reference. Moreover, developers could have a better understanding of their code and the data structure by studying their variables' history.

Current Approach

Currently, time traveling debuggers and other plugins (e.g data visualization plugins) exist that show the history of variables, but they have a multitude of issues. For one, many existing plugins have high overhead, storing information such as stack frames, exceptions, method history, and logs. Chronon Time Travelling Debugger [1] is one such plugin with these limitations. Java Tutor is another tool visualizing the variables' states. However, although tools like Chronon and Java Tutor do a great job in code visualization, it is overwhelming to explain the stack frames of the variables. FootPrint would offer a simple and easy tool just storing variables' information. Furthermore, Chronon also saves execution paths and time stamps for the program, increasing the memory needs. Chronon is not unique in these features; many existing plugins store this information including JIVE [2] and UndoDB [3]. JIVE is a data visualization plugin for Java for Eclipse that also allows the user to see a history of their variables. UndoDB is a time traveling debugger for C/C++ that also tracks memory and register states. All of these solutions are limited by their memory usage, storing superfluous information other than variable states that FootPrint does not.

Related works tend to take the following approaches: recording, using counters, or tracing. Recorders like Chronon, UndoDB, iReplay, etc., record the state of your program throughout the execution, and then allow you to go back and examine the states [4]. These have seemed to have higher overhead than other approaches since the entire execution is recorded and can require gigabytes of logs per day [5]. Kendo uses performance counters, which allow for a more low-level insight on program behavior in order to reproduce bugs for multithreaded programs without recording execution. However, Kendo seems to focus on threading issues such as lock acquisition, race conditions, etc [5]. There is also the use of tracing in gdb's debugger and QIRA and strace. However, single stepping through the program using ptrace like gdb's debugger results in very high overhead [4].

Further, storing all the stack frames, exceptions, method history, etc would make the debugging tools slow in speed. For example, Chronon's overhead can reach >200x for well-optimized Java code [4]. In personal use of these plugins, we have found them to be unintuitive and unreasonably complicated for what we are trying to achieve with them. Finally, one critical issue with current solutions is that many are expensive and are targeted towards corporations rather than individual users. Chronon, for example, has a free open source version, but their Personal plan is a subscription costing $60 a year [1]. UndoDB, on the other hand, has no option for individuals to use or purchase [3]. JIVE and other free plugins exist, however, many still suffer from the issues listed above.

Our Approach

FootPrint allows the user to set watchpoints on variables they wish to track. For each variable that we are tracking, FootPrint will only store changes that happened to it. Consider this scenario:

```
Line 1    int sum = 0 ⟵---------- set a watchpoint here
Line 2    for (int i = 0; i < 6; i++) { ⟵--- set a watchpoint here
Line 3         sum += i;
Line 4    }
Line 5    System.out.println(sum);
```

The user wants to see how the variables sum and i change throughout the for loop. He or she will set a watch point on sum and i to track the changes that will happen to them. Alternatively, a user could proceed to any point in the program after line 5 and would still be able to look back at the histories of sum and i anytime. FootPrint records the different values that sum and i were previously assigned to create the following output for their histories:

```
History:
        sum → 0 → 1 → 3 → 6 → 10 → 15
        line → 1 →  3 → 3 → 3 → 3 →   3

        i     →  0 → 1  → 2 → 3 → 4 → 5
        line →  2 →  2 → 2 →  2 → 2 → 2
```

Notice how sum = 0 only gets recorded once but in the code, sum actually took on the value “0” twice (once during initialization and once during i = 0) but we would only record when the variable changes. This prevents us from storing duplicate information. Furthermore, our approach uses less memory because we are only storing information about specific variables (specifically, values and the line number where it was changed) as opposed to extra data such as stack frames, exceptions, method history, and logs of the entire program like current approaches. Our users are only given information that are relevant to them and are able to get a quick summary of how things change throughout the program.

Architecture & Implementation

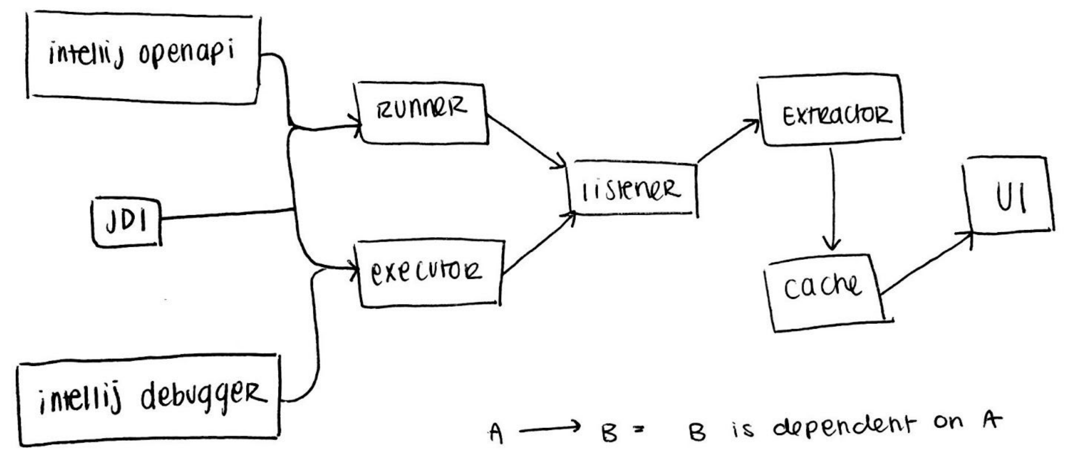


Figure 1.2. Architecture diagram illustrating the major components of FootPrint and their dependencies

FootPrint will use three resources: IntelliJ's openapi, IntelliJ's debugger interface, and JDI (Java Debug Interface). Using these, we will create our own Runner and Executor classes in order to run IntelliJ's built-in debugger through our plugin. Then, we will create a Listener class that will watch for changes in the program's variables. Our Extractor class will extract the contents of these variables when the Listener indicates a change. This data will be stored in a Cache that we implement. The contents of this cache will then be displayed through the UI.

IntelliJ's run actions for programs are as follows. First, a user will select a run configuration and an executor. From IntelliJ's documentation [10], a run configuration allows users to run certain types of external processes such as scripts. From IntelliJ's documentation [11], an executor is a specific way of executing any possible run configuration. Intellij has three built in executors: Run, Debug, and Run with Coverage. For Footprint, we want it to work with any run configuration, so we have created our own executor, extending the DefaultDebugExecutor. A program runner, which will actually execute the process, is then chosen from all registered program runners by asking whether they can run the given run profile with the given executor ID. Because we want access to the internal state of the running debug process and because we have created a custom executor, a custom program runner was necessary. Finally, ProgramRunner.execute() is called, starting the process. Thus, our custom executor and runner live on top of the existing architecture for IntelliJ's debugger.

The listener class that we plan to create will work as follows. The custom runner will, as part of the initialization of the debug process, register with the virtual machine a number of breakpoints, either field or line, that will, when triggered, notify the listener class. The listener class will then notify the extractor class, passing along the necessary information including the stack frame, and will then resume the program. All of this will occur without the knowledge of the user, and will not affect the user's defined breakpoints.

In terms of API's and libraries, FootPrint will use JDI, IntelliJ's XDebugger, and IntelliJ's openapi. XDebugger, the built in underlying debug library used by Intellij, and openapi, Intellij's api for the editor, are used when implementing the runner and executor. Java Debug Interface (JDI) is used actually extract the variable data.

GUI's

Figure 1.3 Interactive GUI

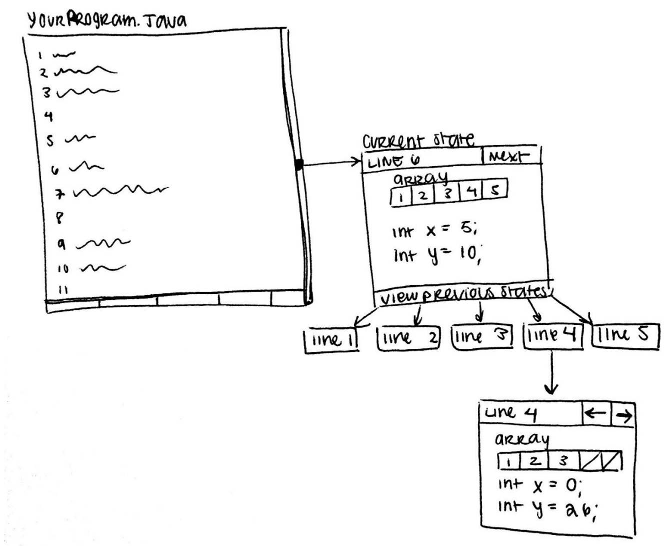
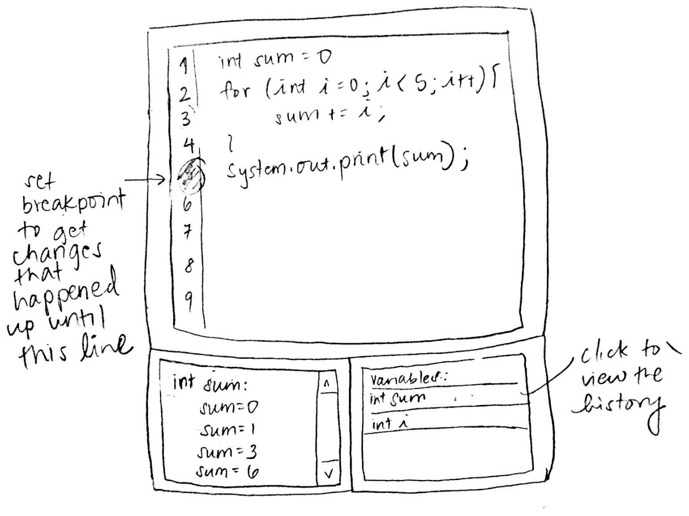


Figure 1.4 Text-based GUI


int sum = 0
for (int i = 0; i < 5; i++){
sum += i;
}
System.out.print(sum);
set breakpoint to get changes that happened up until this line
int sum:
sum=0
sum=1
sum=3
sum=6
Variables:
int sum
int i
click to view the history

Currently we have 2 potential UI's. The first is more interactive and integrated with IntelliJ's built in debugger. One would simply have the option to step back in the debugger and view variable states like normal. The second is more text-based, the user would essentially select a variable and the UI would display its history altogether.

Impact

We think Footprint could be useful for Java programmers using IntelliJ (the IDE we plan to implement FootPrint for). Even if someone doesn't overstep the debugger, being able to see variable history would eliminate the need to try to reproduce bugs. Viewing variable history would also allow for users to better understand what is going on in their code at a high level perspective. However, for beginner programmers who are more likely to overstep the debugger or who may not have as much intuition as senior programmers and would like to easily view what happened to their variables, FootPrint would be especially useful. Overall, this means less time and frustration spent on debugging. We can measure the performance of FootPrint by measuring and comparing memory use and user feedback. Our goal is to create a lightweight debugger that will be more efficient and use less memory that the current solutions, while still being a powerful solution.

Risks and Rewards

We expect the biggest challenge of this project is detecting when a variable is changed up until the breakpoint set by the user. In order to extract information about a variable, Java debuggers need an "event", such as a breakpoint, to pause the program's execution and get the information from the variable [6]. If we want to track the changes that happened, we would need to set our own breakpoints at every line before the user's breakpoint in order to monitor those changes. However, doing so would defeat the purpose of creating a fast and lightweight plugin. We have been advised to use watchpoints instead, as a way to track specific variables which sounds much more efficient. We just have to figure out how to incorporate watchpoints into our implementation now.

In addition, there is the question of how to display this data. We would like to integrate data visualization into FootPrint and have it be more interactive like the built-in debugger, however, this would be a challenge to implement. On the other hand, we can have a text-based display, which would be easier to implement but less user-friendly. For now, we will focus on the backend implementation (the actual extraction and storage of variable history) and decide on how to display this info later on to minimize risk.

Despite this, we think that our solution is feasible since IntelliJ does provide open APIs that allows us to interact with its built in debugger [7] . We will take advantage of this and build our solution upon existing resources. To minimize risks, we will implement basic features first and then add more as time permits.

Cost and Time

Since FootPrint will build on some of the functionalities from IntelliJ's debugger (like extracting variable states), we expect to be able to finish this project within 6 weeks. The first four weeks will be spent on building and testing the backend of extracting and caching information while the remaining two weeks will be spent on building and testing a user interface.

Checks for Success

The midterm to measure FootPrint project is to finish the building process of FootPrint's backend. At this point, we will have fully implemented and tested the storage of debugging information. The final exam to check for success will be after the UI design of FootPrint and the launch of the project. We will do experiments with the users for feedbacks to further improve FootPrint.

Scientific and practical interest

There are three different types of debuggers: cyclic debuggers, record-replay debuggers, and reverse debuggers. As Engblom (2012) pointed out, reverse debugging has been discussed since the very beginning of computer programming. However, it was long ignored because of the difficulty in its implementations, such as the time management and reconstruction approach [8]. Since we are pursuing a similar feature of reverse debugging from the user end but with a simple and light-weighted backend, we came out the idea of extracting the information from debuggers and build a tool to track the footprint of variables [9].

Experimental Methodology

Run FootPrint, Chronon, and JIVE on an open source program in debugging mode. Set breakpoints in various places in the program (i.e. beginning, middle, end) and record the memory consumption of each reverse debugger plugins. The plugin that takes the least memory and has the fastest execution overall is the one with the best performance.

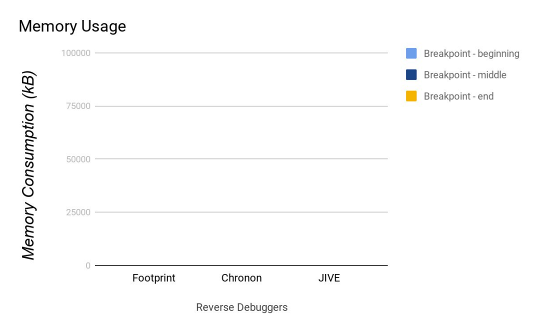


Figure 1.1. Potential memory usage graph

User-Friendliness Experiment

We will gather a small sample of developers (e.g. our friends) who have never been exposed to Chronon and JIVE and have them test Footprint, Choronon, and JIVE on the same program and see which one they think offers the most intuitive and understandable UI through a user survey. Essentially just letting people we know test out FootPrint and seeing if they think our UI is reasonable. Through the results of the survey, we can see if FootPrint achieves its goal of being more user-friendly than current solutions on the market. We will ask them for their overall ranking of the plugins from most favorite to least favorite. Then we will ask in more detail: Was there anything that was difficult to view? Did you encounter any bugs, were the controls intuitive, if you had any issues was the user manual sufficient to solve your issues, would you use this product again? We plan to refine our questions when the experiment approaches and we have a better idea of what FootPrint will be capable of.

Team Assignments

Extractor: Eric, Derek
Cache: Audrey, Hang

Week-by-week schedule

Week 5: Architecture and Implementation plan
We will outline the architecture plan for Footprint. We will decide on a UI and give a mockup. Furthermore, since we are extracting info from the built-in debugger, we will research the architecture of that. We will also decide on what data structures to use for our backend.

Week 6: User manual + begin implementation
User manual - ReadMe.MD (Sections: About, How to Download, How to Use)
Implementation - Finish and test module that will extract variable states from the debugger. Finish and test module that will store the variable states.

Week 7: Build and Test
We will complete and do testing of the UI. Further, we will create a user survey to determine usability and stability of FootPrint. At this point, FootPrint's basic features should be usable.

Week 8: Initial Results
We will compile results of user feedback survey and previous test results. Based on these, we will add to the test suite and fix any apparent bugs. We will run the outlined experiments to gauge the memory performance of FootPrint compared to other products.

Week 9: Draft Final Report
At this point FootPrint should be complete and fully tested. We may add more example programs to run FootPrint on, but otherwise coding should be complete.

Week 10: Finalize Project Report
Finish final commits on Git and finalize the report.

References

[1] Chronon Time Travelling Debugger. (n.d.). Retrieved January 27, 2019, from http://chrononsystems.com/products/chronon-time-travelling-debugger

[2] About JIVE. (n.d.). Retrieved, January 27, 2019, from https://cse.buffalo.edu/jive/

[3] UndoDB. (2019). Retrieved January 27, 2019, from https://undo.io/products/undodb/

[4] Mozilla. (n.d.). Mozilla/rr. Retrieved February 4, 2019, from https://github.com/mozilla/rr/wiki/Related-work

[5] Olszewski, Marek & Ansel, Jason & Amarasinghe, Saman. (2009). Kendo: Efficient Deterministic Multithreading in Software. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS. 44. 97-108. 10.1145/1508244.1508256.

[6] Package com.sun.jdi.event. (2018, October 06). Retrieved from https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/com/sun/jdi/event/package-summary.html

[7] JetBrains. (2019, February 04). JetBrains/intellij-community. Retrieved from https://github.com/JetBrains/intellij-community/tree/master/platform/editor-ui-api/src/com/intellij

[8] Engblom, J. (2012, September). A review of reverse debugging. In System, Software, SoC and Silicon Debug Conference (S4D), 2012 (pp. 1-6). IEEE.

[9] Run Configurations. (2018, September 14). Retrieved February 4, 2019, from https://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations.html

[10] Execution. (2017, October 30). Retrieved February 4, 2019, from https://www.jetbrains.org/intellij/sdk/docs/basics/run_configurations/run_configuration_executio n.html