Live probes for free

Jean-Baptiste Döderlein[a], Tijs van der Storm[b], and Riemer van Rozen[b]

a ENS Rennes, Bruz, France

b CWI, Amsterdam, The Netherlands

Abstract *Context* In his presentation "Inventing on Principles", Bret Victor demonstrates a live code editor: by specifying input values for a function, we can observe in real time the values taken by the variables during execution, as the code is written. This information is often obtained using a language designed for live programming or by instrumentation of a specific runtime. *Inquiry* ...*Approach* In this paper we propose to exploit the capabilities of debuggers to obtain the data needed to design a live code editor. *Knowledge* ...*Grounding* ...*Importance* ...

ACM CCS 2012

- **General and reference** → *Computing standards, RFCs and guidelines*;
- **Applied computing** → **Publishing**;

Keywords programming journal, paper formatting, submission preparation

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission Social Coding, General-purpose programming

1 Introduction

2 Problem Overview

3 Using the Debugger ?

3.1 Stack Recording

```
1 //foo(3)
2 int foo(int n) {
3   int i = 0;
4   while (i < n) {
5     i++;
6   }
7   return i;
8 }
```

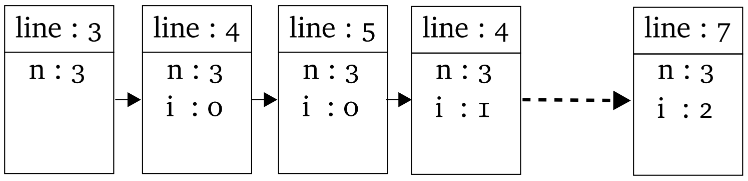


Figure 1 Stack Recording Example

In order to generate the data needed to probe a variable, we need to be able to retrieve the state of the variable during execution. This information is contained in the stackframe at the time of execution. However, we need to add spatial and temporal information to this: we need to associate each stackframe state with the location in the code at which it was retrieved, and we also need to know the order in which these stackframes were retrieved.

In this paper, we introduce a structure for representing this data: a *stack recording*. A stack record represents the different states of the stack frame during the execution of a method. The stack record is presented as a chain of recorded stack frames, to which information has been added about the source code's location and height in the stack. This representation allows us to maintain a link between the spatial location (the reference to the source code) and the temporal location (the order of these stackframes) of the execution. This representation has several advantages: it is easy to construct from debugger information, and it applies to most programming languages. Figure 1 shows an example of a stack recording. On the left is a C function and on the right is the stack recording of the execution of `foo(3)`. Each rectangle represents the state of the stackframe during execution, with the source code location (here simplified to the line number).

3.2 Keep Alive Agent

A live environment must be able to react to two different events: a change in the code or a change in the test/input data. If the code changes, we need to re-execute the code, and in the case of a compiled language, we need to compile the new code first. If the data changes, we need to be able to re-execute the programme with the new

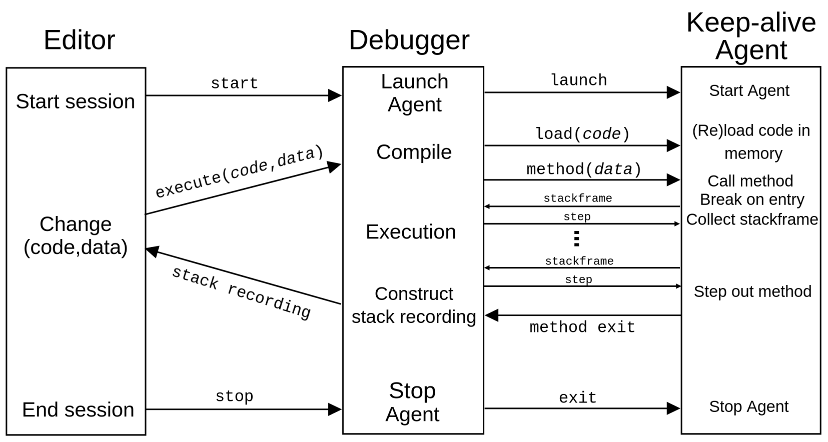


Figure 2 Keep Alive Agent

data. In addition, it is necessary to keep compilation and execution times low enough to maintain an interactive experience with the user.

To meet these constraints, we propose to use the debugger on an intermediate program, a *keep-alive agent*. This program keeps the debugger alive between executions and code changes to reduce initialisation times. The general operation of a keep-alive agent is shown in Figure 2 :

- At the start of the session, the debugger is started on the agent. Once initialised, the agent is paused.
- The target code is loaded into the agent and a breakpoint is set at the input of the target method.
- Execution of the target method triggers the breakpoint at the input of the method, and execution continues step by step to build the *stack recording*.

4 Live Probes in Java with JDI

We developed a Java backend using JDI, a debugging interface for Java, to implement the concepts discussed in the previous section. The keep-alive agent includes a method for loading classes into the JVM.

When executing the target method, the arguments are created in the client JVM and then passed to the debugger JVM using the `mirrorOf` and `newInstance` methods from the reflection API. The target method is invoked using `invokeMethod`. To handle events from the JDI when breakpoints are set, a dedicated thread is used to prevent deadlocks.

If modifications are needed in the code of the target method, we employ the `redefineClasses` method. This method allows changing the content of a class loaded in the JVM using an array of byte codes. However, it has a limitation: it only works if the class signature remains unchanged. If the class structure is modified, restarting the debugger is necessary.

5 Generalizing Live Probes with Debugger Adapter Protocol

The Debug Adapter Protocol (DAP), developed by Microsoft, is a standard method of communicating with a programming language debugger. It is compatible with various editors, including VS Code, and provides a unified interface for all programming languages.

By exploiting this protocol, we have created a new language parametric backend, which we have implemented for the C, Python and Java programming languages. These languages are chosen to cover both compiled and interpreted languages. This backend offers an interface common to all three languages, which includes methods for starting and initialising the debugging server, loading and reloading code in the debugger, and executing a method while performing stack recording.

These methods allow us to carry out stack recording independently of the chosen language and facilitate the future implementation of live programming interfaces.

The implementation for each language includes a keep-alive agent and code to communicate with the debugger and keep-alive agent to provide the interface functionality. These methods depend on both the implementation of the debugging server and the keep-alive agent. The debug server initialisation parameters are specific to each language.

The way in which the code is loaded also depends on the language. For interpreted languages such as Python, the code can be interpreted at runtime and then loaded into the debugger's memory. For Java, we have extended the ClassLoader to add and modify classpaths and classes at runtime in the keep-alive agent. For C, the code is loaded into shared libraries that can be added and reloaded during execution.

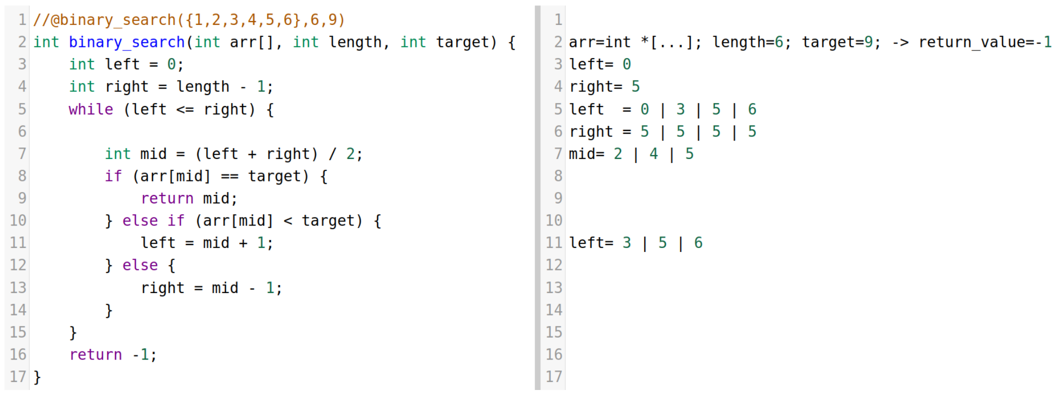
Method execution for Java and Python is different to that used in C. For Java and Python, calling methods directly from the debugger does not trigger breakpoints for these languages. To remedy this, execution must be initiated from the agent and not by a debugger command. To do this, the Python and Java agents have fields for referencing a method and its arguments; when this information is entered, the agent starts execution. For C, as is the case in the backend using JDI, the method call is made from the debugger console.

6 Evaluation

6.1 Demo : A Minimal Live Programming Environment

We have developed a dynamic programming environment that incorporates live probes through Java, C, and Python debug servers. The user interface consists of a web page that interacts with a local server. In Figure 3, we present a screenshot depicting a session with a binary_search function in C. On the left-hand side, there is a code editor (based on CodeMirror 5), while on the right, a panel showcases live probes.

Whenever changes are made in the code editor, the code is sent to the server, which then checks for parseability and syntax errors. Additionally, if the code contains a comment starting with "@", followed by a function call, the server attempts to create



```c
//@binary_search({1,2,3,4,5,6},6,9)
int binary_search(int arr[], int length, int target) {
    int left = 0;
    int right = length - 1;
    while (left <= right) {

        int mid = (left + right) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

```

arr=int *[...]; length=6; target=9; -> return_value=-1
left= 0
right= 5
left  = 0 | 3 | 5 | 6
right = 5 | 5 | 5 | 5
mid= 2 | 4 | 5



left= 3 | 5 | 6
```

Figure 3 Demo of the live programming environment for C.

a stack recording of that function, along with the provided parameters. This stack recording is subsequently processed through a parser to generate live probes, offering the following functionalities:

- Displaying input arguments and return values for each function call.
- Showing the value of each variable definition or assignment. In cases where the assignment is within a loop or called multiple times, all the different values are displayed.
- Presenting the comparison values for each loop (for and while).

6.2 Performance

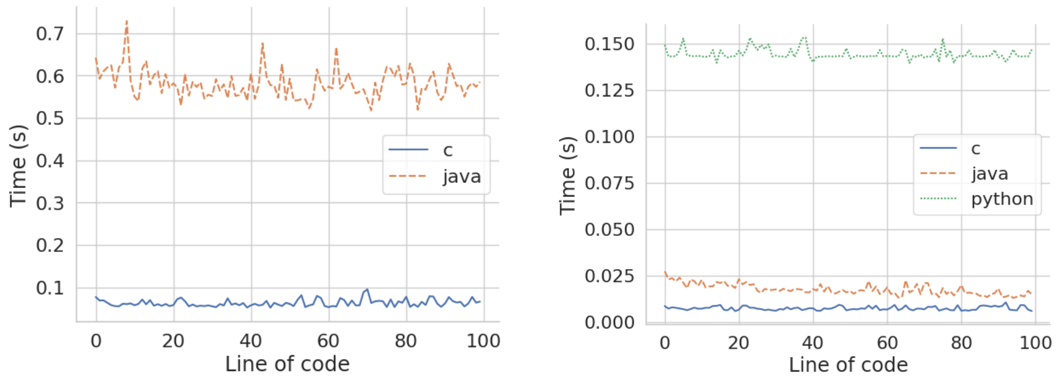
In the context of a live programming environment, it is essential to have short response times after user interactions. We therefore measured the performance of our approach in response to various questions:

- How long does it take to compile and to load the code into the debugger, depending on the size of the programme?
- How long does it take to run the program, depending on the number of stack frames needed to make the stack recording?
- What is the average performance during a live programming session?

6.2.1 How long does it take to compile and to load the code into the debugger, depending on the size of the programme?

In our evaluation, we assessed the time required to compile and load code into the debugger memory for Python, C, and Java. To accomplish this, we compiled and loaded programs ranging from 5 to 100 lines of code. The summarized results can be found in Figure 4.

In Sub-figure 4a, we present the compilation time for C and Java based on the number of lines of code. The compilation process was executed from the command line using javac and gcc. The compilation times remain nearly constant, regardless of the number of lines of code, with an average of 34.6 ms for Java and 8.5 ms for C.



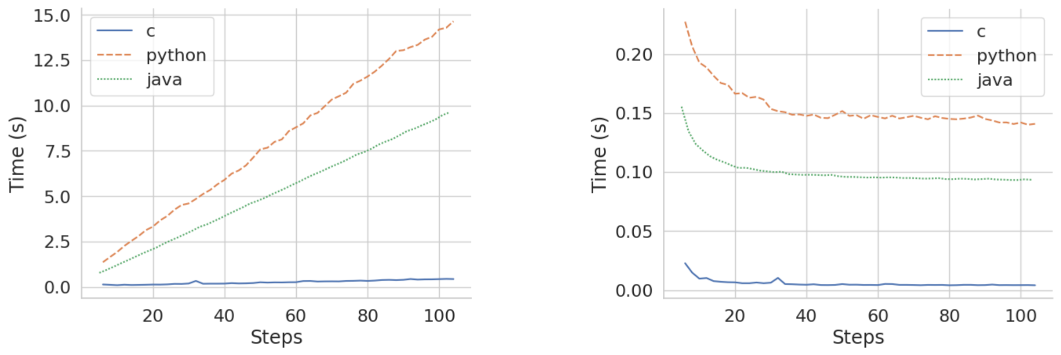
(a) Compile code time

(b) Load code time

Figure 4 Compile and load code time depending on the number of lines of code.

Moving to Sub-figure 4b, we depict the loading time in the debugger for C, Java, and Python as a function of the number of lines of code. The data shows that the loading time remains constant concerning the number of lines of code. On average, Python takes 3 ms, Java takes 2.8 ms, and C takes 1 ms for loading.

6.2.2 How long does it need to run the program, depending on the number of steps to make the stack recording ?



(a) Total execution time

(b) Execution time per step

Figure 5 Execution time in seconds depending on the number of steps.

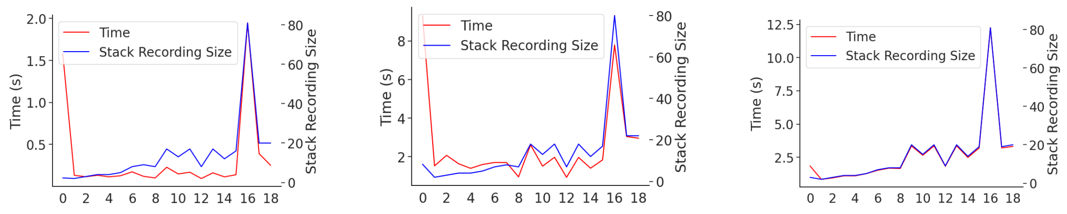
Subsequently, we proceeded to evaluate the execution time concerning the number of steps in the debugger. Figure 5a presents the total time taken based on the number of steps during stack recording, while Figure 5b demonstrates the time per step as a function of the number of steps. Notably, the execution time exhibits a linear relationship with the number of steps in the method.

6.2.3 What is the average performance during a live programming session?

To evaluate the performance of our approach in the general context of a live programming session, we measured the times taken by the different stages of the interaction loop:

	One time	Each Iteration		
Language	Initialization	Compile	Load Code	Execute
C	1.24	0.067	0.0098	0.193
Java	5.92	0.57	0.015	1.28
Python	0.639	0	0.144	3.26

Table 1 Average time in seconds for each step of the live programming session.



(a) C **(b)** Java **(c)** Python

Figure 6 Execution time and Stackrecording size for binary search scenario.

- Agent initialisation time, which occurs once at the start of the session.
- The compilation time, if there is a compilation stage.
- The time taken to load the code into the debugger.
- Program execution time.

For the 3 languages we have implemented, we have performed the measurements for the execution of a binary search function 7. For each language, the agent was initialized, then the code was compiled, loaded into the debugger and executed (with the array [1,2,3,4,5,6] and target 9) 100 times. Each execution generate a stack recording of approximately 20 stack frames. The results are shown in table 1.

7 Related Work

Example-Based Live Programming for Everyone[3] : Use GraalVM/Truffle to get live information from the code of multiple languages.

Example Centric Programming[1] : Use BeanShell(custom JVM) to get live information. Prototype for Java in Eclipse.

Usable Live Programming[2] : New language for live programming(Ying Yang) with incremental compilation. Live programming environment for this language. Use source location to relate execution and code(=> almost like stack recording that link stackframe and code location)

Scalable Omniscient Debugging[4] : Omniscient debugging with a lot of data. Use a lot of memory to store all the data. In this paper they record almost everything, we only record the stackframe.

8 Conclusion

References

[1] Jonathan Edwards. "Example Centric Programming". In: *SIGPLAN Not.* 39.12 (Dec. 2004), pages 84–91. ISSN: 0362-1340. DOI: 10.1145/1052883.1052894. URL: https://doi.org/10.1145/1052883.1052894.

[2] Sean McDirmid. "Usable Live Programming". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 53–62. ISBN: 9781450324724. DOI: 10.1145/2509578.2509585. URL: https://doi.org/10.1145/2509578.2509585.

[3] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. "Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pages 1–17. ISBN: 9781450381789. DOI: 10.1145/3426428.3426919. URL: https://doi.org/10.1145/3426428.3426919.

[4] Guillaume Pothier, Éric Tanter, and José Piquer. "Scalable Omniscient Debugging". In: *SIGPLAN Not.* 42.10 (Oct. 2007), pages 535–552. ISSN: 0362-1340. DOI: 10.1145/1297105.1297067. URL: https://doi.org/10.1145/1297105.1297067.

Appendix

```c
int binary_search(int arr[], int length, int target) {
   int left = 0;
   int right = length - 1;
   while (left <= right) {

         int mid = (left + right) / 2;
         if (arr[mid] == target) {
               return mid;
         } else if (arr[mid] < target) {
               left = mid + 1;
         } else {
               right = mid - 1;
         }
   }
   return 0;
}
```

```python
def binary_search(arr, target):
      left = 0
      right = len(arr) - 1

      while left <= right:
            mid = (left + right) // 2

            if arr[mid] == target:
                  return mid
            elif arr[mid] < target:
                  left = mid + 1
            else:
                  right = mid - 1

      return 0
```

```java
public class BinarySearch {
      public static int binarySearch(int[] array, int key){
            int low = 0;
            int high = array.length - 1;
            while (low <= high) {
                  int mid = (low + high) / 2;
                  int value = array[mid];
                  if (value < key) {
                        low = mid + 1;
                  } else if (value > key) {
                        high = mid - 1;
                  } else {
                        return mid;
                  }
            }
            return 0;
      }
}
```

Figure 7 Binary Search in C, Python and Java