Position Specification in Program Traces and Potential Applications in Semi-automated Debugging

Kazutaka Maruyama* and Minoru Terada
Dept. of Mechano-Informatics
University of Tokyo

1 Introduction

In debugging programs, we do actions of two classes: examining variables, such as "print" in GDB, and moving the control point, such as "step" or "break". Moving the control point is very tiresome task and incidental to examining variables, so we would like to automate or semi-automate this. Doing so using the execution control facilities of current debuggers is impossible, since they are based on static information in source codes, but states of programs being debugged (called "debuggees") are dynamic. Thus we need to specify points of execution in program traces to simplify our debugging process.

In this paper, by re-executing deterministic C programs[1], we propose a mechanism of moving control to the target points of execution and some applications of it. The implementation with GCC and GDB is also included.

2 Positions in the Execution

```
(1) while(i < a){
(2)   i += b;
(3) }
```

If we execute the program above in a file, `test.c`, line (2) is executed repeatedly. When we aim at the third appearance of line (2) and we set a breakpoint at the line represented as (test.c:2), the execution stops there over and over. We call the statically unique point in the source code, such as (test.c:2), *location*.

Therefore, we introduce a counter, `timestamp`, which is updated when the same location is executed so that each breakpoint hit can be distinguished by using the pair of location and timestamp. In figure 1, (file:line, timestamp) represents the point of execution in the program trace, so we call this *position*. This new notion allows us to solve the problems noted in the introduction.

2.1 Dynamic Breakpoint

A breakpoint of debuggers are set at a location and the execution also stops at positions which are not

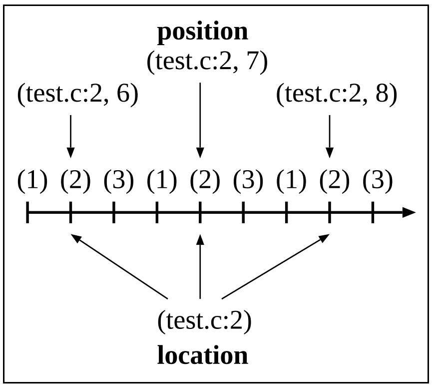

Figure 1: Location and positions.

our target. Now, we introduce *dynamic breakpoint* which are set at a position not a location. This can be implemented by using existing debuggers: (1) we advance the execution until the timestamp becomes the value of the target position, and then (2) continue until the location of the position by using normal breakpoint facility. At this point both timestamp and location match the intended position.

3 Applications of Specifying Positions

If we build a runtime which supports position information, we can automate the following debugging procedures. These are simple, but it is important to be able to automate them by using the computing power which is more than enough.

3.1 Marking positions

When debugging programs running in long time, such as traversing tree structures in game programming, it is useful to mark the position where all things are correct till the time in order to go back there later. It is a dynamic breakpoint itself and we can use comment strings instead of a number as its identifier.

3.2 Reverse Watchpoint

It is hard to locate bugs as overrunning the area of arrays or `malloc`-ed one, because their results

*kazutaka@acm.org
[1] We are trying to handle also C++ programs.

are manifested much later. We usually use watchpoint facility of debuggers, but, generally, too many writes to the destroyed area occur, the execution stops by the watchpoint over and over, we cannot find the real write of destruction easily. However, we know by experience that the write closest in time to the manifestation of the bug is.

We can implement "reverse watchpoint" facility which advances the execution to the write by the following procedure: (pass 1) set a watchpoint [2] to the target area and re-execute the debuggee to get a list of positions where the watchpoint traps occur, and (pass 2) re-execute it again to advance the control to the last write.

3.3 Binary Search Method for Locating Bugs

Timestamp value, which a part of position information, roughly shows the point in the program trace and can be used as a key of binary search. We can walk around in the trace and get the timestamp value of the time when some conditions become false, for example, data structures of double-linked lists become invalid. If the maximum value of timestamp is n, the maximum times of evaluating the condition is $\log n$, then it is no more than 32 times when timestamp has 32 bit length.

4 Implementation of Position

Timestamp has to be updated at the following: (1) entrances and exits of function calls, (2) `return` statements, (3) `goto` statements, and (4) entrance of loops. To implement this portablly, we insert codes updating timestamp into debuggees in the intermediate code level of GCC. We modified the parser, some actions of it, and some macros.

5 Processing Time Measurements

5.1 Overhead of Timestamp System

The speed of debuggees slows down by added codes for updating timestamp. We measure it in case of gawk-2.15.6 and show the result in table 1.

5.2 Elapsed Time of Reverse Watchpoint Operation

Since the elapsed time of reverse watchpoint operation depends on the number of stops by watchpoint in pass 1, we measure the time in different cases of the number (table 2) [3]. The time grows linearly.

[2] GDB has this facility. We use it for portable implementation.

[3] Celeron–400MHz was used for the experiment.

Table 1: Overhead of timestamp system.

GCC options	Celeron 400MHz (linux)	Alpha 500MHz (linux)	SPARC station20 (SunOS)
-g -O (sec.)	2.57	4.64	17.44
-g -O -pg2 (sec.)	3.78	6.34	24.40
factor	1.47	1.37	1.40

Table 2: Elapsed time of reverse watchpoint operation.

trap of watchpoint (times)	elapsed time (sec.)
511	6.56
1011	11.94
1511	17.58
2011	23.46

6 Related Works

Boothe[1] implemented a bi-directional debugger with some counters. All the counters do not need for our position, so overhead of updating them is bigger, around 1.95 times against under 1.5 times of us.

Ducassé[2] implemented the execution control by designating the conditions of breakpoints in events, such as function calls, loop entrances, and so on. However, designating the conditions does not guarantee that the debuggee stops only one position. It suits manual debugging methodology.

7 Conclusions

We proposed the concept of "positions" as a basis of semi-automated debugging, implemented it into GCC portably, and also showed three applications of it.

Please download and try our GCC and wrapper of GDB for using reverse watchpoint. It is available at the following URL: http://www.sanpo.t.u-tokyo.ac.jp/~kazutaka/gcc/

References

[1] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 299–310, 2000.

[2] Mireille Ducassé. Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events. In *Rapport de recherche INRIA RR-3489*, 1998.