# Querypoint : Moving Backwards on Wrong Values in the Buggy Execution

Salman Mirghasemi
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
salman.mirghasemi@epfl.ch

John J. Barton
IBM Research - Almaden
bartonjj@us.ibm.com

Claude Petitpierre
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
claude.petitpierre@epfl.ch

## ABSTRACT

As developers debug, they often have to seek the origins of wrong values they see in their debugger. This search must be performed backwards in time since the code causing the wrong value is executed before the wrong value appears. Therefore, locating the origin of wrong values with breakpoint- or log- based debuggers demands persistence and significant experience.

*Querypoint*, is a Firefox plugin that enhances the popular Firebug JavaScript debugger with a new, practical feature called *lastChange*. *lastChange* automatically locates the last point at which a variable or an object property has been changed. Starting from a program suspended on a breakpoint, the *lastChange* algorithm applies queries to the live program during re-execution, recording the call stack and limited program state each time the property value changes. When the program halts again on the breakpoint, it shows the call stack and program state at the last change point. To evaluate the usability and effectiveness of *Querypoint* we studied four experienced JavaScript developers applying the tool to two test cases.

## Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; D.2.6 [**Programming Environments**]: Integrated environments

## General Terms

Algorithms, Human Factors, Languages

## Keywords

Debugging, Locating Defects, Querypoint, LastChange, Breakpoint, Watchpoint, Logging

## 1. INTRODUCTION

According to [5], developers spend about fifty percent of their time on debugging. To fix a bug, developers typically reproduce and monitor the buggy execution several times to understand the program's unexpected behavior. Trial-and-error, guess-work, and analyzing complicated data make debugging difficult and time-consuming. Thus enhanced debugging operations are highly needed to save time, reduce development costs and improve software quality.

In a common strategy for locating defects, the developer starts from the bug symptoms and traces the execution backwards, moving from a point in the program execution where a value appears to be incorrect back to the point where that value was set. Two conventional approaches, breakpoint-based and log-based debugging, require tedious steps to select the data to be collected, collect these data and, then analyzing the results.

Locating the origin of wrong values becomes even harder when developers deal with weakly-typed dynamic languages such as JavaScript. First, due to weakly-typed nature of these languages the search space that the developer should consider is considerably larger than with traditional languages such as Java and C#. Second, the dynamics features of these languages limit the value of traditional compile-time program analysis.

*Querypoint*, is a Firefox plugin that enhances the popular Firebug JavaScript debugger with a new, practical feature called *lastChange* [8]. *lastChange* makes it possible to locate the origin of a wrong value by making queries about the running program. This feature allows the developer to navigate through dynamic data dependency without requiring a full trace, unlike other interfaces based on dynamic slicing. The proposed technique builds on the existing breakpoint approach and it does not require any special environment to create identical, instruction by instruction, re-executions. *Querypoint* also provides a mechanism to automate the bug reproduction, and a novel user interface that summarizes the data of the investigated execution points and the collected results.

## 2. RELATED WORK

Most tools developed to enhance developers' navigation on a buggy execution can be classified in two main groups: replay-based and logging-based. Replay-based approaches capture limited data during execution and replay the buggy execution to reach past points. In contrast, logging-based approaches collect enough data during execution to relieve the developer from re-executing.

Two replay-based debuggers, bdb [3] and reverse watchpoint [7], rely on deterministic executions and employ a step counter to locate the requested point from the beginning of execution. These tools incur two to four times runtime overhead.

Among logging-based debuggers are *omniscient* debuggers (e.g., ODB [6], TOD [9], and WhyLine [4]) and time-travelling debuggers (e.g., Nirvana [1]). Logging-based debuggers suffer from different issues. First, the recording phase is time expensive (20-120 times) and it must be repeated in case of changes in the program. Second, the execution log cannot fully replace the live execution. There are other aspects of execution (e.g., program user interface, file system, database tables, etc.) that are also important in debugging and are not available to the developer in logging-based debuggers. Third, querying collected data (e.g., to restore the program state at a certain point) may not be efficient enough to debug realistic programs.

*Querypoint* resembles the operational model of replay-based debuggers and the query approach of logging-based debuggers. However, contrary to other replay-based debuggers, which require exactly the same re-executions (deterministic executions), *Querypoint* only requires the *reproducibility* of the bug, so that a test case reproduces the bug and a way to halt the execution reliably after the reproduction is available. Unlike logging-based debuggers *Querypoint* selectively collects data which significantly reduces runtime overhead incurred by logging.

## 3. QUERYPOINT

Let us illustrate the *Querypoint* functionality by a simple example. This example demonstrates a buggy JavaScript code in a HTML page (Figure 1). This page contains a button (line 40) that shows the value of `myObject.myProperty`. When the user clicks on the button, the `onClick` function (line 13) is called. This function increases the value of `myObject.myProperty` by one (line 15) and calls the `updateButton` function that updates the button's text to the new value (line 22). When the page is loaded for the first time, the button shows `1` as the initial value of `myObject.myProperty`. In practice when the user clicks on the button, `0` appears instead of `2`: there is a bug.

By browsing through the code,the developer determines that the value displayed on the button is set at line 22. Since the displayed value is incorrect they know the bug occurred before the program hit this line. To start debugging, the developer sets a breakpoint on line 22. Once the button is clicked, the execution is paused at line 22. Figure 2(a) shows the Firebug debugger while the execution is paused. Firebug has several panels (e.g., HTML, CSS, Script, DOM, etc.) each one demonstrating one aspect of the Web page. The Script panel on the left side contains the list of all loaded source files and regular debugging facilities such as setting breakpoints and stepping. On the right of the script panel, the Watch panel shows the program state that displays the scopes and the variable values. In our case, the `myObject.myProperty` value at the paused point is `0`. We expected this value to be `2`.

To apply the backward search strategy to locate defects, the developer first needs to know the origin of the wrong value. Then they muse use breakpoints, and look through the code to find all possible places where `myObject.myProperty` might get a new value and set a breakpoint at these locations. However, an object and property can be accessed and changed through different names and methods. There is no simple way to identify these aliases or even their total

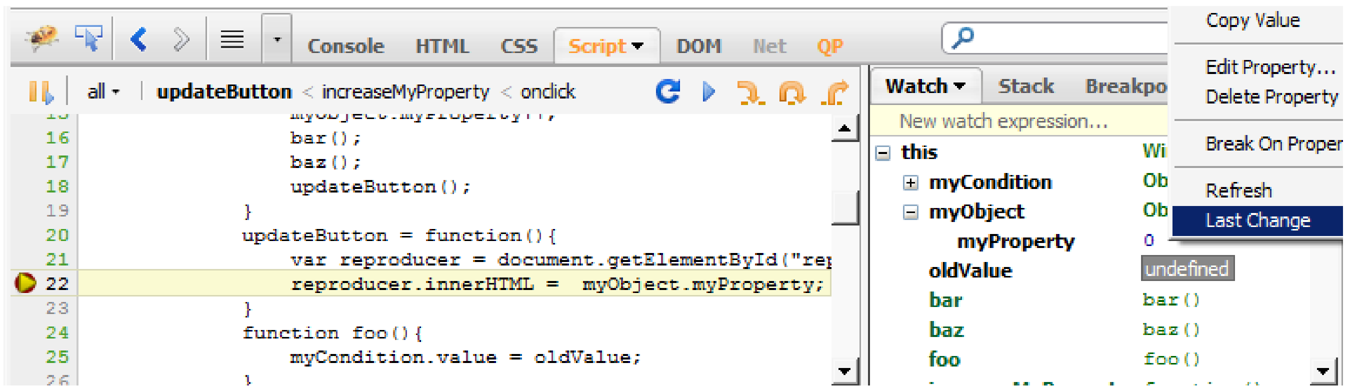
```
1 <html>
...
5    <script type="text/javascript">
6      myObject = {myProperty : 1};
7      myCondition = {value : 1};
...
13     function onClick(){
14       foo();
15       myObject.myProperty++;
16       bar();
17       ...
18       updateButton();
19     }
20     function updateButton(){
21       var myParagraph =
             document.getElementById("myButton");
22       myButton.innerHTML = myObject.myProperty;
23     }
24     function foo(){
25         myCondition.value = oldValue;
26     }
27     function bar(){
28       if (!myCondition.value)
29           myObject.myProperty = 0;
30     }
31   </script>
...
40   <button id="myButton" onclick="onClick()">
41       1
42   </button>
43 </html>
```

**Figure 1: A Web page containing JavaScript code. Some lines not related to our paper have been elided.**

number. The developer is limited to make a good guess and to set breakpoints on lines where the property seems to be changed, to re-execute the program and to examine the state looking for values that may lead to the incorrect value observed at line 22. All this work must be repeated if new aliases are discovered or if some information related to the buggy result was missed while suspended on one of the breakpoints.

In our proposal, we have added a high-level function in the debugger, *lastChange*, that provides the answer without tedious manual effort from the developer. By right clicking on `myObject.myProperty` in the Watch panel, the developer can run the *lastChange* command (Figure 2(a)). The debugger re-executes the program and halts again at the breakpoint on line 22. However, it shows a new panel, called QP, centered on the source at line 29 (Figure 2(b)), the point of *lastChange*. To the right, the TraceData panel shows the values of properties of the program state when it passed through line 29. These two panels resemble the Script and Watch panels, but they show data collected by the debugger at one execution point that is now in the past: these data are *traces* or *logs* collected during the re-execution.

Looking at line 29, the developer concludes that something is wrong with `myCondition.value` which leads to the execution of line 29. The developer then examines `myCondition.value` and noticed that it is `undefined`. The next step the developer performs is to know when this property got this value. To do so, they run the *lastChange* command on `myCondition.value` from this point. The debugger re-executes the program and breaks again on line 22, analyzes the queries and shows the developer line 25-the place `oldValue` is assigned to `myCondition.value`. If the developer asks for *lastChange* on `oldValue`, the debugger can notify

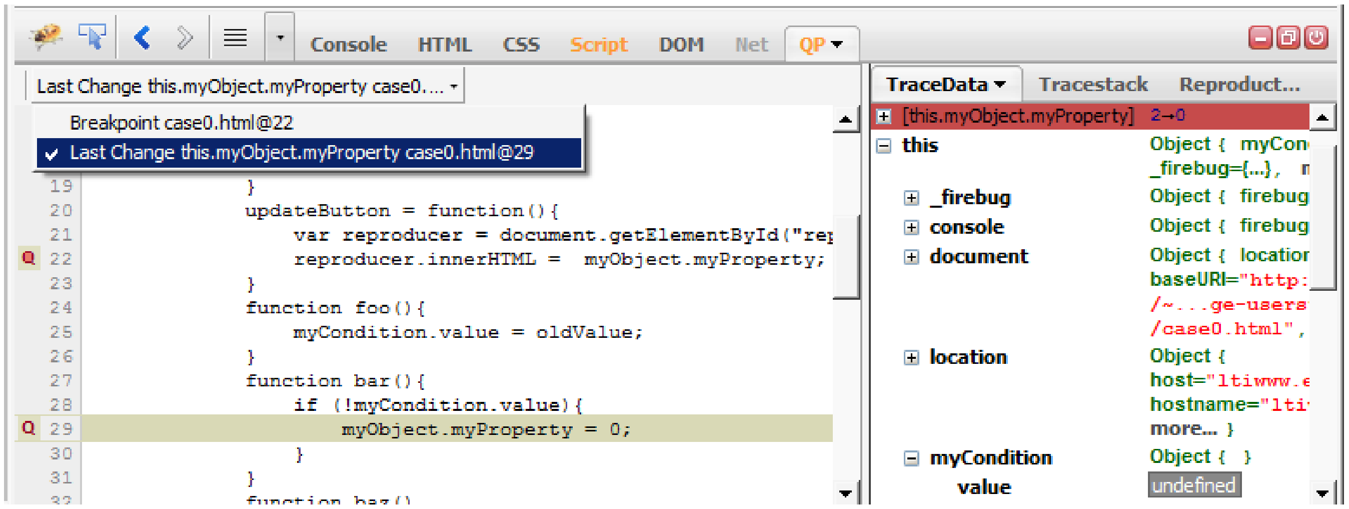


```
all  updateButton < increaseMyProperty < onclick
```

```
16              bar();
17              baz();
18              updateButton();
19          }
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
```

```
this              Wi
  myCondition     Ob
  myObject        Ob
    myProperty    0
  oldValue        undefined
  bar             bar()
  baz             baz()
  foo             foo()
```

(a) A screen shot of the Firebug debugger while running the example code from Fig. 1. The Script panel is selected; it gives access to all loaded source files and allows breakpoints to be set on lines. In this figure, the execution is paused at line 22 by a regular breakpoint. The Watch panel on the right shows the program state at the breakpoint. The developer can query *lastChange* on `myObject.myProperty` by right-clicking on the value of `myProperty`.

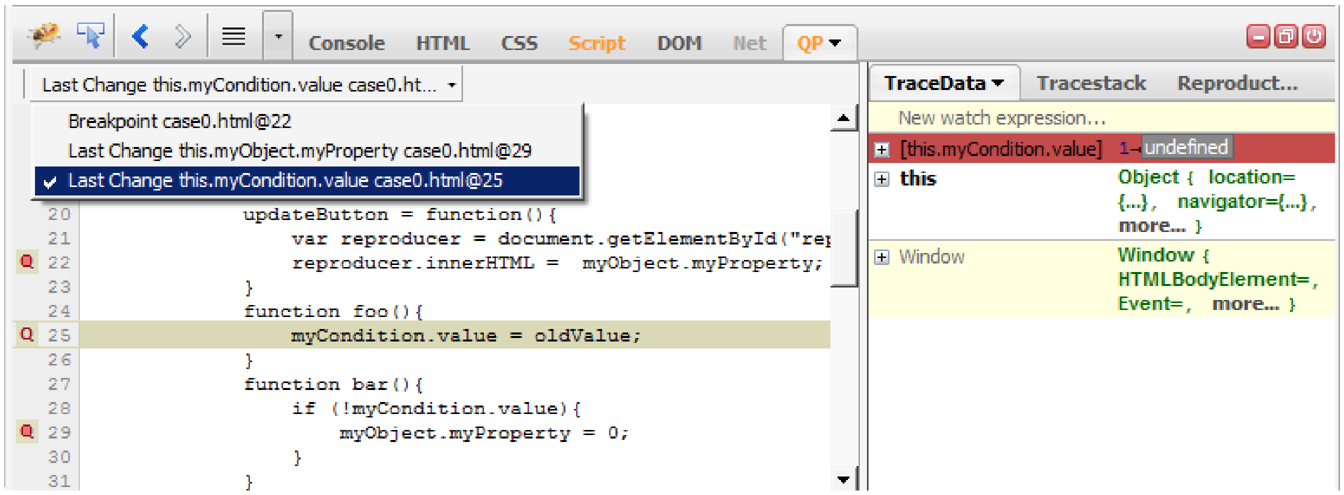


```
Last Change this.myObject.myProperty case0....
  Breakpoint case0.html@22
✓ Last Change this.myObject.myProperty case0.html@29
```

```
19          }
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
26          }
27          function bar(){
28              if (!myCondition.value){
29                  myObject.myProperty = 0;
30              }
31          }
```

```
[this.myObject.myProperty]  2→0
this              Object { myCon
                  _firebug={...}, r
  _firebug        Object { firebug
  console         Object { firebug
  document        Object { locatior
                  baseURI="http:
                  /~...ge-users
                  /case0.html",
  location        Object {
                  host="1tiwww.e
                  hostname="1ti
                  more... }
  myCondition     Object { }
    value         undefined
```

(b) The result of *lastChange* query for `myObject.myProperty`. The left panel, QP, shows the source code at the point of *lastChange*; The right panel, TraceData, shows the collected data at the point.



```
Last Change this.myCondition.value case0.ht...
  Breakpoint case0.html@22
  Last Change this.myObject.myProperty case0.html@29
✓ Last Change this.myCondition.value case0.html@25
```

```
20          updateButton = function(){
21              var reproducer = document.getElementById("rep
22              reproducer.innerHTML =  myObject.myProperty;
23          }
24          function foo(){
25              myCondition.value = oldValue;
26          }
27          function bar(){
28              if (!myCondition.value){
29                  myObject.myProperty = 0;
30              }
31          }
```

```
[this.myCondition.value]  1→undefined
this              Object { location=
                  {...}, navigator={...},
                  more... }
Window            Window {
                  HTMLBodyElement=,
                  Event=,  more... }
```

(c) The result of *lastChange* query for `myCondition.value`. To evaluate an expression (e.g., oldValue) at this point, the developer can enter the expression in the watch box and after a re-execution the result is available. The opened list on the top of the left panel shows the visited execution points. Clicking on each point in the list shows the corresponding code and data.

**Figure 2: The stages of locating the defect using *lastChange* feature.**

the developer that this variable has never been assigned a value. Now it is clear that the bug occurs because `oldValue` is `undefined` when the execution reaches line 25 (Figure 2(c)).

The developer has examined three points of execution, but set one explicitly. We call the latter breakpoint the *reproduction point*. The second and third points precede the reproduction point in the execution sequence. All three points correspond to the history of the search for the defect. They are available through the debugger's interface. At the top of the left panel in Figure 2(c) there is an opened list that shows all three examined points. Moreover, the source lines related to these points are marked with red **Q** icons.

*Querypoint* needs a test case to reproduce the execution and conditions to correctly recognize the reproduction point. Although both elements can be directly provided by developer, *Querypoint* is also able to automatically create them from the first execution.

To replay an execution, *Querypoint* keeps track of breakpoint and single steps the execution went through. *Querypoint* supports two mechanisms for automatic re-execution: callstack-reproduction and record-replay. In callstack reproduction the function from the bottom frame of the call stack is called with the same parameters. The record-replay execution uses two phases. In the record phase, it stores the initial page url and the events and parameters corresponding to user actions. In the replay phase, it opens the same url and simulates events as if they were user actions.

In addition to the data collected at every change event for identifying the *lastChange* result, *Querypoint* partially stores values in program state. There is a trade-off between the amount of data collected at every change event and the number of re-executions. If the developer asks for some values that have not been stored, *Querypoint* re-executes and collects the requested data.

## 4. USER STUDY

We supplied four experienced Javascript developers with *Querypoint* integrated in an extended Firebug debugger[1]. Following a tutorial and a practice case, we observed them as they applied both conventional breakpoint and *lastChange* on two small programs we provided. The first program, Shapes, calculates the area and perimeter values for a list of shapes. The bug happens when one of the calculated numbers is zero. The second program, Moving Circle, randomly scales and moves a circle in the page. The bug happens once the circle becomes invisible after an exception occurs. This case represent a reproducible non-deterministic execution. The developers were asked to locate the defects that caused these bugs. All four developers successfully applied *lastChange* to the test programs and understood how it could help debugging. To find the defect location with breakpoints, all four users took more steps[2] and more time (Figure 3).

| Programs \ Developers | DEV1 | DEV2 | DEV3 | DEV4 |
|---|---|---|---|---|
| **Shapes** | **3 (85 s)** | 13 (182 s) | 39 (318 s) | **3 (80 s)** |
| **Moving Circle** | 9 (215 s) | **4 (40 s)** | **3 (195 s)** | 12 (234 s) |

**Figure 3: The number of steps (and time in seconds) required before locating a defect, for each test subject and test program. Cells with a white background report values with conventional debugging; Cells with a colored background use *lastChange*.**

## 5. CONCLUSION AND FUTURE WORK

*Querypoint* provides critical information for debugging JavaScript programs: the location and state at the point where a questionable value was assigned. It only requires bug reproducibility and is built on the existing breakpoint technology.

In our next iteration we plan to merge the query and breakpoint results. *Querypoint* shows the results of *lastChange* in a similar but different view from breakpoint debugging. This focuses attention on value changes, but it makes studying control flow more difficult.

[1]http://ltiwww.epfl.ch/~mirghase/lastchange-userstudy

[2]A step is a button push, either single stepping the debugger or running a *lastChange* query.

## 6. REFERENCES

[1] S. Bhansali, W. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, J. Chau. Framework for instruction-level tracing and analysis of program executions. In *International Conference on Virtual Execution Environments(VEE)*, June, 2006.

[2] M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.

[3] B. Boothe. Efficient algorithms for bidirectional debugging. In *Conference on Programming Language Design and Implementation(PLDI)*, June, 2000.

[4] A.J. Ko, and B.A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *30th international conference on Software engineering(ICSE)*, May, 2008.

[5] T.D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits In *28th international conference on Software engineering(ICSE)*, May, 2006.

[6] B. Lewis, and M. Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, 2003.

[7] K. Maruyama, and T. Kazutaka. Debugging with Reverse Watchpoint. In *Proceedings of the Third International Conference on Quality Software*, 2003.

[8] S. Mirghasemi, J.J. Barton, and C. Petitpierre. Debugging by lastChange. Technical Report. EPFL-REPORT-164250, 2011.

[9] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. In *22nd annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(OOPSLA)*, October, 2007.