How Omniscient Debuggers Impact Debugging Behavior

Ruochen Wang
George Mason University
Fairfax, VA, USA
rwang29@gmu.edu

Thomas D. LaToza
George Mason University
Fairfax, VA, USA
tlatoza@gmu.edu

***Abstract*—Debugging is an essential yet often tedious part of the software development process. Omniscient debuggers have long aimed to make debugging easier by recording execution traces, enabling more direct debugging interactions. Although the concept of omniscient debugging has been explored extensively in research, it has seen limited adoption in industry until recently. The emergence of new commercial tools like Replay presents an opportunity to reevaluate the impact of omniscient debugging. In this paper, we conducted a controlled experiment with 20 participants with a commercial omniscient debugger, Replay, and a traditional debugger, Chrome DevTools. We investigated whether the omniscient debugger improved developer productivity and how it influenced debugging behavior. We coded developers' navigation, rerun, and runtime value collection behaviors and summarized their debugging strategies. Our results show that developers with the omniscient debugger were not more successful or faster than those using the traditional debugger. Omniscient debugger users reran the program less, but there was no significant difference in the number of files or functions they explored or the number of runtime values they collected. Omniscient debugger users faced navigation and runtime value collection challenges, which may have hindered their effectiveness. Our results suggest that commercial omniscient debuggers must include more of the high-level support for interacting with traces found in research prototypes to successfully help developers in debugging tasks.**

***Index Terms*—debugging, omniscient debuggers**

I. Introduction

Debugging is the process of identifying, locating, understanding, and fixing defects in software. It is a necessary yet tedious part of the software development process, which often consumes more time than creating the software [1]. Debugging can be difficult for many reasons. Developers often first encounter a bug through its symptom, which might appear as a missing behavior, unintended behavior, or an error message. Different types of symptoms can influence the difficulty of the debugging task. An error message can make debugging easier because the developer will have a clear starting point by searching for the text contained in the error message. From this starting point, the developer must trace back to the root cause. The gap between the symptom and the ultimate cause is described by the concept of cause/effect chasm [2]. A large chasm—due to large spatial, temporal, or data flow gaps—can make debugging more challenging.

One influential approach to tackling the challenges of debugging is the omniscient debugger [3], [4]. Omniscient debuggers record the execution trace of the program and use the trace to help developers debug. Omniscient debuggers can enable developers to step backward, while regular breakpoint debuggers can only step forward. This allows developers to go back when they stepped forward too far without rerunning the program, as well as to trace backward from the symptom of the defect to its cause [5]. Omniscient debuggers may also enable developers to navigate directly to the code responsible for a program output or that responds to an event, saving the developer time to find the code manually. The recorded trace can also be used to help developers answer their questions.

For many years, omniscient debuggers that record every expression during execution remained available only in research prototypes or in systems for small programs suitable for educational use. In 2012, Undo began offering support for omniscient debugging for Linux system programming [6]. In 2021, Replay began offering a commercial omniscient debugger for debugging front-end web applications [7]. In this paper, we examine how such an omniscient debugger may impact how developers debug. More specifically, we examine:

- RQ1. When do omniscient debuggers improve debugging productivity?
- RQ2. How do omniscient debuggers change navigation behavior when debugging?
- RQ3. How do omniscient debuggers change how developers rerun the program when debugging?
- RQ4. How do omniscient debuggers change how developers collect the values of expressions?

We conducted a controlled experiment with 20 participants, comparing debugging behavior between developers using an omniscient debugger (Replay) and a traditional debugger (Chrome DevTools). We coded and compared the navigation, rerun, and view value behaviors of the developers, examined their debugging strategies, and investigated how the tool either supported or failed to support their debugging process. Overall, we found that the use of the omniscient debugger did not significantly improve the productivity of developers. While the omniscient debugger reduced the number of reruns, it did not significantly reduce the effort of navigation or value collection. Our results suggest that, despite the frequent focus on stepping backward as the central interaction enabled by omniscient debuggers, other interactions, such as connecting output to code, may be more important for effectively supporting debugging.

II. Related Work

Our work builds on prior studies examining the process and challenges of debugging, new forms of proposed debugging tools, and prior studies of omniscient debuggers.

A. Debugging theories and challenges

A number of theories and models have conceptualized the work developers do when debugging. According to program slicing [8], developers do not examine programs contiguously, but follow mentally constructed program slices, a subset of the program that could influence the value of an expression. According to information foraging theory [9], [10], developers navigate through the code to find the "prey" (location of the bug) based on its "scent" (e.g., linguistic similarity of function identifiers and the bug report) and face challenges choosing which function to read next. Eisenstadt [2] identified reasons why debugging is difficult, including large temporal or spatial chasms between cause and symptom. Developers spend an average of 35% of their time simply navigating between dependencies, and 46% inspecting task-irrelevant code [11]. According to hypothesis-based theories of debugging [12], [13], developers form and test hypotheses, which are often difficult to conceptualize and incorrect [14]. Debugging has also been described as a process of asking and answering questions about the code and runtime behavior [15], [16].

Many tools have been conceived to help developers debug more effectively. For example, automatic fault localization tools [17] identify statements that may contain the bug, and automatic program repair tools [18] generate patches that fix the bug without human intervention. However, while most developers use the debugging features in IDEs [19], they tend to avoid complex features and primarily use breakpoints [20]. When compared to not using a debugger, developers spend more time debugging and performed more navigation actions when using one, possibly because debuggers are used in more difficult bugs [21]. The same study also found that, in most sessions, developers maintained the same editing behavior regardless of whether they used a debugger. Automated debugging tools may be helpful if they directly provide the correct fix [22]. However, for tools that do not directly provide a fix, the accuracy of the tool may not be as important, as a study [23] found that the rankings of the suggested faulty statements did not impact the performance of developers. Studies have also found that developers prefer human support, such as asking their colleagues for help, over debuggers [24].

B. Omniscient debuggers

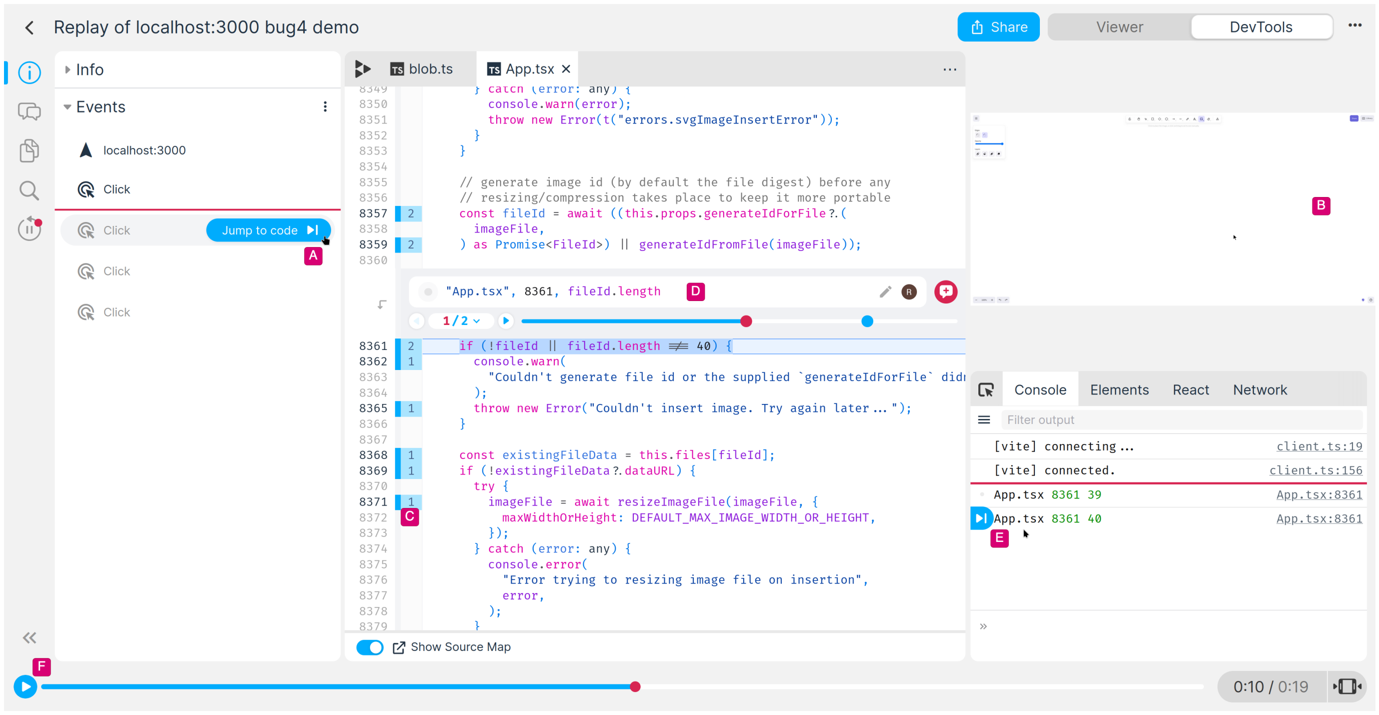
Building on the earlier concept of a reversible debugger [25] introduced by Lieberman in 1995, the concept of an "omniscient debugger" was introduced by Lewis in 2003 and refers to a debugger that allows developers to examine the runtime value of variables at any time by recording every state change in a program [3]. One way to enable developers to interact with this recording is to enable stepping backward to a statement that was executed before the current statement, rewinding the state of the program [3], [6], [7], [25]–[28]. Researchers believe that stepping backward allows developers to trace back from the symptom of the bug to its root cause [3], [26]–[28]. It can also eliminate the need to rerun the program if the breakpoint is set after the faulty statement [3], [27].

However, the ability to step back may not be enough if the chasm between the symptom and the cause is large. Instead of just stepping back sequentially through control flow, some omniscient debuggers support explicitly navigating data and control flow relationships. Using dynamic slicing [29], the Slice Navigator [30] allows developers to step between statements within a slice to directly rewind to the statement that last modified the value of a variable. Object-centric omniscient debuggers [31]–[33] allow developers to track an object to see its changes over time and across method boundaries. Whyline [5] and NuzzleBug [28] allow programmers to directly ask "why" and "why not" questions about program behavior and provide the causal chain of the behavior using dynamic slicing.

Using the recorded trace, some omniscient debuggers support directly navigating to the code responsible for a program output or that responds to an event. Given an output or an event, they identify it in the recorded trace, navigate to the relevant statement, and rewind the program state to that point in the execution. This is thought to reduce manual fault localization because developers can locate the faulty statement with a related program event, which is something that the developer can directly observe [34]. For example, systems allow developers to rewind the execution to when a line of output is printed (ODB [3] and Replay [7]), a graphical element is drawn (ZStep 95 [25]), or a user event occurs (Timelapse [34] and Replay [7]). Starting from the program output, developers can step backward to locate the fault. Alternatively, if the defect happens after a user action, they can jump to that program event and step forward to the faulty statement.

Some omniscient debuggers use queries on the execution trace to help developers test hypotheses and answer questions. Expositor [35] and Time-Traveling Queries [36] allow programmers to write queries on the trace to answer their questions about the code and showed that commonly asked questions during debugging [16] can be expressed using queries [36]. These two omniscient debuggers support the verifying of hypotheses and answering of questions, while other tools also support the forming of a hypothesis or question. Hypothesizer [37] suggests possible hypotheses to developers, querying the trace to find patterns of program behavior that support a hypothesis. It also helps developers test the hypothesis by displaying relevant information extracted from the trace. Because real-world programs can quickly generate vast numbers of state changes, scalability is a major barrier to using omniscient debuggers in practice, and achieving better performance and scalability has been a major focus [38]–[41].

Several user studies have evaluated the productivity of research prototypes built to explore the concept of omniscient debugging. Some of these studies have found productivity benefits. Time-Traveling Queries [36] was found helpful for



```
    } catch (error: any) {
      console.warn(error);
      throw new Error(t("errors.svgImageInsertError"));
    }
  }

  // generate image id (by default the file digest) before any
  // resizing/compression takes place to keep it more portable
  const fileId = await ((this.props.generateIdForFile?.(
    imageFile,
  ) as Promise<FileId>) || generateIdFromFile(imageFile));

  "App.tsx", 8361, fileId.length

  if (!fileId || fileId.length !== 40) {
    console.warn(
      "Couldn't generate file id or the supplied `generateIdForFile` didn
    );
    throw new Error("Couldn't insert image. Try again later...");
  }

  const existingFileData = this.files[fileId];
  if (!existingFileData?.dataURL) {
    try {
      imageFile = await resizeImageFile(imageFile, {
        maxWidthOrHeight: DEFAULT_MAX_IMAGE_WIDTH_OR_HEIGHT,
      });
    } catch (error: any) {
      console.error(
        "Error trying to resizing image file on insertion",
        error,
      );
    }
```

```
[vite] connecting...                client.ts:19
[vite] connected.                   client.ts:156
App.tsx 8361 39                     App.tsx:8361
App.tsx 8361 40                     App.tsx:8361
```

Fig. 1. Replay records the execution trace, showing a list of recorded user events (A), some of which have a corresponding "jump to code". Developers can play back the recording (F) and see the webpage output (B). In the gutter next to each line (C), developers can view a hit count of the times a line has been executed. Developers can insert console log statements (D) to view expressions they want to examine, and Replay automatically reruns the program and updates the console output (E). Developers can also jump from output in the console to the corresponding console log statement (E).

developers to answer program comprehension questions, reducing the number of actions required to answer these questions. The Java Whyline [5] was found to help developers debug faster and more successfully by helping them rely mostly on answering questions, rather than text searches for relevant content. Hypothesizer [37] was found to improve time and success by supporting developers in the process of forming and testing hypotheses, helping developers to have a deeper understanding of the defect. The omniscient debugger for Scratch, NuzzleBug [28], was evaluated in an educational setting. Students were significantly more successful in 1 of 8 tasks, with the biggest benefits for bugs in complex programs.

Other studies have found no productivity benefits of omniscient debuggers. A study of Timelapse [34] found no significant improvement in time or success, in part because less-skilled developers became distracted by Timelapse's features. The Omission finder [42] is an add-on of the omniscient debugger Traceglasses [43] that helps identify execution omission errors (some statements should have been executed but were not). A user study with 24 participants [42] found that the Omission finder only reduces the task time when there is an execution omission error and if the execution trace is long. It also found that developers who used Traceglasses (without Omission finder) had similar task times as those who used a traditional breakpoint debugger.

Together, these findings suggest that the productivity benefits of omniscient debuggers may depend on other factors, such as the type or difficulty of the debugging task or the specific capabilities of the tool. In particular, these studies suggest that the productivity benefits depend on providing higher-level aid to developers, where tools that did not help required developers to examine the execution trace themselves. In this paper, we further explore this question, offering the first user study evaluating a real-world commercial omniscient debugger and examining in detail how omniscient debuggers impact debugging behavior across different types of defects.

III. Replay

Replay is one of the first commercial omniscient debuggers for web applications [7]. Developers debug with Replay by first demonstrating a defect or behavior of interest and recording an execution trace using the Replay browser. Developers then have full access to the execution trace. Replay offers:

- Step backward as well as forward (Figure 2)
- Causal navigation: Developers can see a list of recorded mouse and keyboard events, and jump to the code that responds to an event. They can also jump from console output to the corresponding console log statement in the code. (Figure 1–A, E)
- Live console logs: Developers can insert a new console log statement in the code, and Replay will then automatically simulate rerunning the recorded session with the new console log statements and display the updated output. (Figure 1–D)

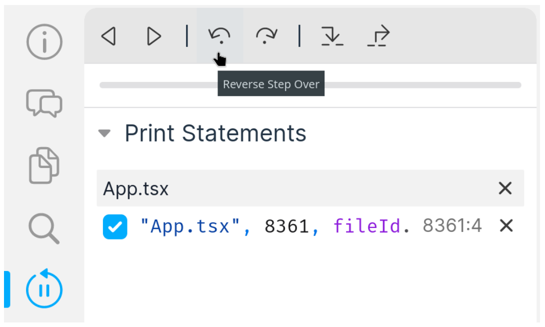
- Hit count: Looking only at the source code, developers can see the number of times each line of code was executed in the gutter. (Figure 1–C)

Based on these features, we formulate the following hypotheses on how Replay may change developer behavior.

H1. Developers visit fewer files and functions, and they focus on the relevant functions faster. (RQ2) Step backward enables tracing from symptoms to causes without exploratory navigation, while causal navigation provides direct links from events to relevant code locations.

H2. Developers rerun the program less, and they rerun with different purposes. (RQ3) This is because the ability to record the execution reduces the need to rerun the program. Live console logs and hit count might also eliminate the need to rerun the program for value inspection or determining if a line of code is executed.

H3. Developers use different debugger features to view the value of expressions and view different types of expressions. (RQ4) Specifically, live console logs may reduce the friction of using print statements since developers no longer need to rerun the program, potentially shifting developers' preferred methods for runtime value collection. Developers can also view more complex expressions with print statements.



```
"App.tsx", 8361, fileId. 8361:4
```

Fig. 2. Developers can control the current position in the execution recording. They can execute the recording backward or forward until hitting a breakpoint (left two buttons), step backward or forward one statement (middle two buttons), and step into the function under the cursor or return from the current function (right two buttons).

IV. Methodology

To investigate the impact of omniscient debuggers on debugging behavior and answer our research questions, we conducted a controlled experiment with 20 participants. Specifically, we compared the use of Replay to Chrome DevTools, a widely used tool for front-end web debugging. Our materials and data are publicly available. [44]

A. Participants

After obtaining IRB approval, we recruited 20 participants from four sources: 10 from a graduate-level software engineering course at our university, 6 from a graduate student mailing list at our university, 1 from LinkedIn, and 3 from personal contacts. All were currently masters students or junior software engineers. We required participants to have at least 7 years of programming experience and experience with React and JavaScript. This threshold was established after early participants with less experience were unable to make progress, leading us to revise our inclusion criteria. Participants ranged in programming experience from 7 to 20 years, with a median of 8 years. Most had professional experience as a software engineer, ranging from 0 to 10 years, with a median of 3.29 years. Participants had from 0 to 4 years experience with React, with a median of 1 year. When asked to estimate the number of lines of JavaScript they have written, 2 participants responded with 100 to 1000 lines, 9 reported 1000 to 10 000 lines, and the rest 9 reported more than 10 000 lines. We did not ask participants about their familiarity with Excalidraw. However, their behavior during the study suggested that all were new to the code base.

B. Tasks

Participants worked on tasks on Excalidraw [45], a virtual whiteboard web application. Excalidraw is a popular open source project on GitHub with 73.1k stars. It uses the React framework, contains 289 TypeScript files, and is approximately 80,000 kLOC. To systematically examine debugging behavior for different types of defects, we inserted 4 defects, each occupying a different quadrant along two key debugging challenge dimensions: the size of the cause-effect chasms and the presence or absence of an error message (Table I). Bugs 1 and 2 were selected from the Excalidraw GitHub issue tracker. Bugs 3 and 4 were manually created. Bug 3 was designed to have a corrupted state introduced early that causes visible failures later. Bug 4 involved incorrect hex string formatting.

We provided relevant background for three of the bugs, when the task was likely too difficult to complete within 40 minutes (Bug 1) or required app-specific terminology (Bugs 2 and 3). For Bug 1, we provided a pointer to the function that rendered the element with the bug. For Bugs 2 and 3, we gave a brief explanation of unfamiliar terms.

TABLE I
Bugs used in the study

Bug	Chasm	Symptom	Description
1	large	no error message	Cannot clear name when editing frame name
2	small	no error message	Incorrect alignment when binding
3	large	error message	Cannot add arrow to library
4	small	error message	Cannot insert image

C. Procedure

The experiment was conducted remotely on Zoom. Participants completed tasks by remotely controlling the experimenter's laptop via Zoom. Participants first completed the consent form and the demographics survey. The demographics survey asked participants about their programming experiences, familiarity of the programming language and framework used in the tasks and their tool usage habits when debugging web applications. After the survey, they were shown tutorials of both Replay and Chrome DevTools. For each tool, participants first read a tutorial with text and screenshots introducing the key features, and then completed a warm-up task. Participants then completed two debugging tasks, one with Chrome DevTools and one with Replay. They were allowed to explore or modify the code using VSCode and to search on the web. Participants were asked to think aloud while working.

We divided the four tasks into two sets. Set A included Bug 1 and Bug 4, and Set B included Bug 2 and Bug 3. Each participant got either Set A or Set B. In this way, the two bugs that each participant received had different symptoms and sizes of the cause-effect chasm. We counterbalanced the order of the two tasks within the set and the order they use the debugging tool, creating 8 conditions. Participants were assigned to these 8 conditions sequentially based on the time of their study. If a participant was excluded because they were unable to make progress, the next participant was assigned to the excluded participant's condition.

After completing the tasks, participants took part in a semi-structured interview. Questions focused on participants' strategies for the two tasks, the challenges they encountered, the ways in which tool support did or did not help address the challenges, and the features they found helpful.

Midway through the study, the interface of Replay was unexpectedly updated. Specifically, the ability to add breakpoints and the resume and rewind buttons were removed (the left- and right-pointing triangle buttons in Figure 2). This may have changed the way participants rewind to a specific line.

D. Analysis

To examine how debugging behavior may change with use of an omniscient debugger, we examined debugging behavior by qualitatively coding our screen recording data. We focused on three categories of codes: *navigation*, *view value*, and *rerun*. To construct the code book, the first author selected the first 4 screen recordings to build the initial code book. We then refined the codebook, and the two authors applied the code book iteratively to the same screen recordings and compared the results. After coding 7 screen recordings (2 hours in total), we performed an inter-rater reliability test and reached a Cohen's Kappa of 75.25% for navigation codes, 81.7% for view value codes, and 77.94% for rerun codes, indicating substantial to almost perfect agreement [46]. The first author then coded all the recordings with the final version of the code book. During this process, we found two new types of navigation action codes that we did not encounter when constructing the code book, so we added them to the code book after the inter-rater reliability test. We argue that adding these two new actions will not affect the reliability of the code book, so we did not perform a new inter-rater reliability test.

Navigation codes capture a change to the currently visible top-level function or declaration, or switching to another editor or debugging tool. For each code, we recorded the new file, new function, and the navigation action that the developer performed to reach the new function. We categorized navigation actions into navigation actions that "use the debugger", such as stepping into a function or using Replay's jump to code feature, and those that "do not use the debugger" such as scrolling, clicking on a file, and editor features like going to the definition of a function (Table II).

View value codes capture actions to view the runtime value of an expression (Table III). This includes hovering on an identifier in the debugger to see its current value as well as adding a print statement and rerunning to see the value in the console. We did not code instances when the debugger displays runtime values without any intervention from the developer. For example, Chrome always displayed the values of the variables currently in scope. We did not create codes unless the developer clicks on it to expand its fields.

TABLE II
NAVIGATION ACTIONS WITH OR WITHOUT THE DEBUGGER

Action	Description	Use debugger?
Scroll	Scroll mouse wheel to function	no
Go to	Go to the definition of a function	no
Click on file	Click on a file in the file explorer	no
File search	Perform search within one file	no
Global search	Click on a global search result	no
Switch tab	Switch to another open tab	no
Switch tool	Go to another tool	no
Step	Step into/out of the current function	yes
Hit breakpoint	Breakpoint hit, jumping to a function	yes
Jump to code	Jump from event or console log to code in Replay	yes
Console error	Click warning/error link in console	yes
Debugger links*	Click on call stack, or breakpoint item in the debugger	yes
Pause on exception*	Use the "pause on caught/uncaught exception" feature in chrome	yes

*: added after the inter-rater reliability test was done

We created a *rerun* code every time the developer reproduced the bug in the web browser and recorded the purpose for the rerun. The purpose was determined by the actions before or after the rerun. We identified 4 purposes for which participants rerun, and they are listed in Table IV.

TABLE III
ACTIONS TO VIEW THE VALUE OF AN EXPRESSION

View action	Description
Pop up	Hover over variable to see value in a pop up
Expand locals	Expand a variable in the local variable view
Chrome watch	Add a watch in Chrome
Print statement	Add a print statement
Console	Type an expression in the console

V. Results

We provide the answers to our research questions in this section. Before examining the impact of Replay on productivity and debugging behavior, we first analyze how participants utilized Replay's three features that was captured in our codes (Table V). We can see that nearly all participants used the three features. However, as the following analyses reveal, feature usage did not translate to improved debugging productivity.

TABLE IV
PURPOSES FOR RERUNNING THE PROGRAM

Purpose	Criteria
Initial rerun	Reproduce in Chrome or Replay before any investigation efforts
Observe behavior change	If the previous action is code behavior modification
Collect runtime value	If the participant collected runtime value during the rerun, or added a print statement before the rerun
Determine if code is run	If the previous action is adding a breakpoint
Other	If does not fall under the purposes above

TABLE V
REPLAY FEATURE USAGE PATTERNS

Feature	Participants who used	# uses per session
Jump to code	17	2 (1–5)
Record/Rerun	19	2 (1–2)
Print statement	17	2 (1–6)

Note: number of uses is reported as “median (interquartile range)”

A. RQ1. *When do omniscient debuggers improve debugging productivity?*

We compared the success rate and the task time for the four bugs when developers use Chrome or Replay (Table VI). Because the task times of some conditions are not normally distributed (Shapiro–Wilk test $p < 0.05$), we used Mann-Whitney U tests to compare the time distributions. Overall, we found that, for both task time and success rate, there were no significant differences between developers using Chrome and those using Replay.

To examine if Replay helps developers find the key function related to the bug, we defined a milestone function (MS1) for Bugs 2, 3, and 4 and coded the time participants reached this function. Since we already provided participants the function that rendered the element in Bug 1, we exclude this bug for analysis of MS1. For participants still working to reach MS1 at the end of the session, we coded their time as the full session time (40 minutes). We found that Replay did not help users reach MS1 significantly faster (Figure 3).

TABLE VI
SUCCESS RATES AND TASK TIME IN MINUTES PER BUG

	Tool	Success	Time (minutes)	U test $p =$
Bug 1	Chrome Replay	2/5 1/5	40.0 (35.9-40.0) 40.0 (40.0-40.0)	0.239
Bug 2	Chrome Replay	4/5 4/5	11.9 (11.1-38.2) 24.2 (11.8-34.6)	0.917
Bug 3	Chrome Replay	3/5 2/5	34.6 (27.4-40.0) 40.0 (35.5-40.0)	0.106
Bug 4	Chrome Replay	2/5 2/5	40.0 (19.3-40.0) 40.0 (29.2-40.0)	0.504

Note: Time is reported as “median (interquartile range)”

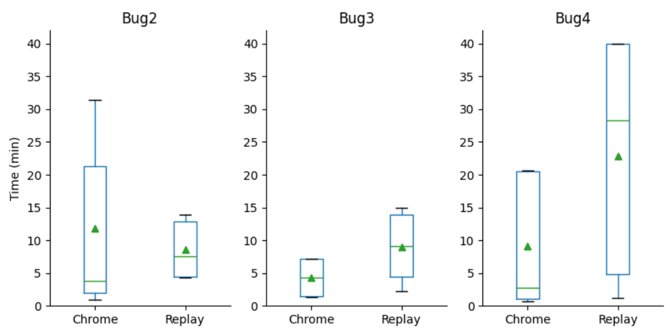


Fig. 3. Box plot of the time participants reached MS1

B. RQ2. *How do omniscient debuggers change navigation behavior when debugging?*

Developers need to navigate in the code to identify the bug. With features such as causal navigation, we hypothesized that developers would visit fewer files and functions, and would focus on the relevant files and functions faster. However, this hypothesis is not supported by our observations.

1) The number of files and functions developers visit is similar: We calculated both the number of files and functions that the developer visited in each debugging session. There were no significant differences between Chrome and Replay in either overall (Mann-Whitney U test $p = 1.0$ and 0.68, respectively), nor for individual defects (Table VII).

TABLE VII
NUMBER OF FILES AND FUNCTIONS VISITED

Bug	Tool	Files visited		Functions visited	
Bug1	Chrome Replay	3 (2-4) 3 (2-3)	$p = 0.74$	8 (5-24) 9 (7-10)	$p = 0.83$
Bug2	Chrome Replay	7 (3-7) 3 (3-8)	$p = 0.92$	14 (4-17) 5 (4-15)	$p = 0.92$
Bug3	Chrome Replay	8 (7-9) 11 (8-11)	$p = 0.39$	19 (12-20) 21 (16-22)	$p = 0.42$
Bug4	Chrome Replay	4 (2-6) 5 (4-7)	$p = 0.60$	6 (4-20) 10 (10-11)	$p = 0.68$

Note: numbers are reported as median (interquartile range)

2) The number of navigation actions is also similar, but developers used different types of navigation actions: We counted the occurrences of different navigation actions from the coded data. First, we compared the total number of navigation actions during debugging with Chrome and Replay (Figure 4-a) and found no significant difference (Mann-Whitney U test $p = 0.409$). We then examined the number of navigation actions which “uses the debugger” or “does not use the debugger” (Figure 4-b,c). The median number of navigation actions that “uses the debugger” was 3x higher for Replay users than for Chrome users, and this difference was significant (Mann-Whitney U test $p = 0.0439$). There is no significant difference for navigation actions that “do not use the debugger” (Mann-Whitney U test $p = 0.776$).

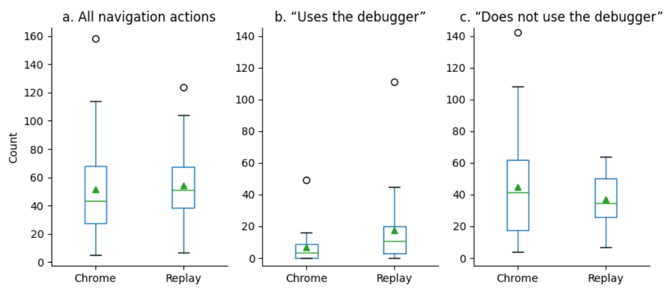


Fig. 4. Number of navigation actions

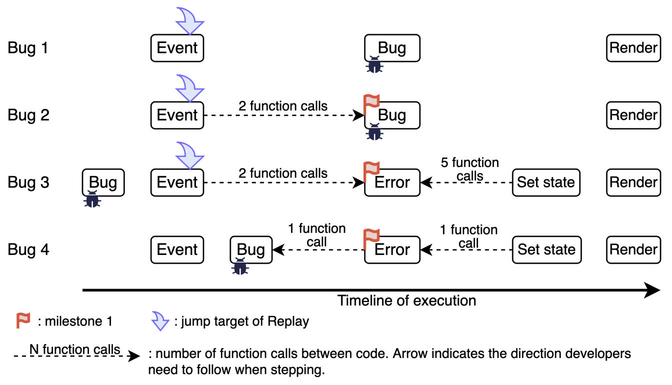


Fig. 5. Temporal relationships between different code segments: the event handler ("Event"), the statement with the bug ("Bug"), the statement that throws the exception ("Error"), the code that modifies the current error message in the app state ("Set state"), and the code that renders the output ("Render"). In Bug 2, the bug is in the MS1 function, while in Bugs 3 and 4, participants need to further navigate to find the bug after they reach MS1.

3) Replay's "jump to code" feature did not help participants reach MS1 faster: In Replay, "jump to code" takes the developer directly from a user event or console output (but not graphical output) to the code that responds to the event or prints the output. We expected this to give the developer a clear starting point to examine the code, enabling them to navigate toward the cause of the defect. We expected it to be particularly useful when working with unfamiliar code, as in our experiment. However, in RQ1, we observed that Replay users did not reach the MS1 function faster than Chrome users (Figure 3). Although two-thirds of the Replay participants initially used "jump to code", fewer than one-third of those participants succeeded in navigating from the click event to MS1—and two participants never reached MS1 at all.

We found three barriers that kept "jump to code" from reducing the navigation burden. First, the "jump to code" target was before, not after, the function they need to navigate to, so it could not be used with other omniscient debugging features, like stepping back. The ability to step back is an important feature of omniscient debuggers and can be used to trace back from the symptom of the bug to its root cause [3]. However, Replay's "jump to code" target for user events is typically before MS1 (Figure 5). Therefore, they could not use the step back feature. The jump target for program output (console or graphical) is typically after the defect. While Replay supports jumping from console output to code, because there was no console output for the four bugs, it could not be used. Had Replay supported jumping from graphical output to the code like ZStep 95 [25] or Whyline [5], developers might have been able to use the step back feature to locate the fault faster.

Second, it was difficult for participants to rewind the trace to a specific line when they needed to inspect the application state at that line. They needed to add a breakpoint at the line, and then use the rewind/resume buttons (Figure 2) to hit the breakpoint. Three participants (P3, P6, P9) incorrectly clicked the play recording button (Figure 1–F) instead of rewind/resume, so the recording did not pause as expected. Another way to rewind is to select "Fast forward to line X" in the line's context menu. However, this was not discoverable.

In Bugs 2 and 3, this usability issue reduced the effectiveness of the "jump to code" feature. Replay displayed the event for a context menu click that triggered the bug. After participants clicked the "jump to code" button for this event, Replay rewound the recoding to an onClick event handler (line 103 in Figure 6), which registered another call back that executes the action containing the bug (line 104). Although lines 103 and 104 were adjacent, they were not executed sequentially, so developers could not navigate to line 104 using step over or step into. Because it was challenging to rewind the recording directly to line 104, participants could not successfully navigate to MS1 and had to revert to traditional debugging techniques. While Replay was able to help jump to the event handler, participants were not able to locate the function containing the bug.

Third, certain user interactions were not recorded by Replay. In Bug 4, the bug was triggered after an image insertion event. Participants first clicked the "insert image" button on the web page, then selected an image in the operating system window. Although the first click on the button was recorded, the interaction in the OS window, specifically the "Open" button that inserted the image into the web page was not recorded. Therefore, the "jump to code" feature was not helpful.

4) Replay did not help navigate causal dependencies: Bug 3 had a large chasm, because the defect was caused by a corrupted state that was modified in a different function. As shown in Figure 5, once developers identified the corrupted state (at the "Error" location), they needed to navigate to the "Bug" location where the state was set. However, since the corrupted value was not passed as a parameter, it was not obvious where the modification occurred.

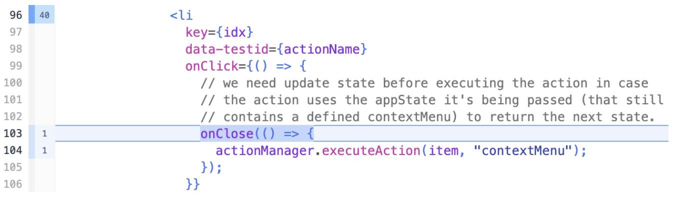

```
96  40           <li
97                 key={idx}
98                 data-testid={actionName}
99                 onClick={() => {
100                  // we need update state before executing the action in case
101                  // the action uses the appState it's being passed (that still
102                  // contains a defined contextMenu) to return the next state.
103  1               onClose(() => {
104  1                 actionManager.executeAction(item, "contextMenu");
105                  });
106                }}
```

Fig. 6. The jump to code target of Bugs 2 and 3

Replay did not help navigate this chasm. Although its step back feature allows developers to go backward in time, it offers no guidance on *where* to step to locate the code that modified the relevant state. In contrast, research prototypes such as Whyline [5], dynamic slicers, and object-centric omniscient debuggers [31] may offer more effective support. These tools directly address questions like "Why does this variable has this value?", or "Where was the value set?" by performing dynamic slicing or tracking the changes of objects. Unlike Replay, they not only allow developers to go backward in time, but also guide them toward the relevant code locations.

C. RQ3. How do omniscient debuggers change how developers rerun the program when debugging?

In traditional debugging, developers often insert breakpoints or print statements to inspect program states. If they miss a critical state or need to re-examine a previous one, they must restart the program and reproduce the scenario, which can be time-consuming. With Replay, developers do not need to rerun to inspect program states. Therefore, we hypothesized that Replay users would rerun the program less frequently, and that Replay and Chrome users would rerun the program with different purposes. Our data supported these hypotheses.

We counted the number of reruns in each debugging session. Across all 20 Chrome sessions, the median number of reruns is 10, while the median is 2 for Replay sessions. This difference is significant (Mann-Whitney U test $p = 0.000019$). We also compared the number of reruns for each bug (Table VIII), and the number of reruns of Chrome users is significantly higher for Bugs 3 and 4 ($p < 0.05$).

TABLE VIII
NUMBER OF RERUNS

Bug	Tool	# reruns	U test $p =$
Bug1	Chrome Replay	15 (9-18) 3 (2-4)	0.075
Bug2	Chrome Replay	5 (3-5) 2 (1-3)	0.070
Bug3	Chrome Replay	12 (10-17) 2 (2-2)	0.045
Bug4	Chrome Replay	10 (10-15) 2 (2-2)	0.020

Note: numbers are reported as median (interquartile range)

Replay and Chrome users differed in their purposes for rerunning the application (Figure 7). Replay users used rerun for two specific purposes. Developers first used the "initial" purpose to record the bug in the Replay browser, enabling them to then reproduce it in its entirety in the Replay browser. In Bugs 3 and 4, which were caused by invalid inputs, 5 out of 10 participants also reran the program with a valid input so that they could compare the runtime states in the two scenarios. Replay users also used rerun to "observe behavior change". This is when they test their hypothesis for a potential fix by modifying the code and rerunning in the browser.

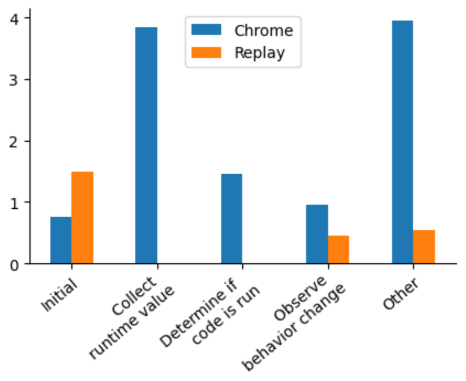


Fig. 7. Average number of reruns by purpose

Chrome users reran the program for a wider variety of purposes. In addition to the two purposes of Replay users, they also reran to collect runtime values and to determine if code is executed in a specific scenario. When the developer added a print statement before the rerun, or viewed runtime values during the rerun, we coded it as "collect runtime values". If they did not collect any runtime value, but they added a breakpoint before rerunning, we coded it as "determine if code is run". This typically happened when developers suspect a function is related to the bug, and they want to test if it is executed when they trigger the bug. Replay users do not need to rerun to find out if code is executed because the hit count feature shows the developer how many times a line is executed. About 35% of the reruns involve collecting runtime values, and 13% have the "determining if code is run" purpose.

In summary, we found that developers reran the program less when using Replay because they do not need to rerun to collect runtime values or determine if code is run.

D. RQ4. How do omniscient debuggers change how developers view the values of expressions?

One of the key expected benefits of omniscient debuggers is enabling developers to collect the value of a new expression without having to rerun the program. We examined the impact of this feature on how developers choose to collect runtime values. Specifically, we examined the number of expressions viewed, the debugger features developers used to view them, the types of expression developers viewed, and the barriers that hindered developers during this process.

We define a value collection action as a deliberate step a developer takes to retrieve the runtime value of an expression. This includes hovering over an expression to view its value in a pop up, as well as the act of adding a print statement and reproducing the bug (if needed). We first compared the number of value collection actions and the number of unique expressions collected by developers and found no significant difference (Table IX).

For both Chrome and Replay users, the hover pop up was the most common way to collect runtime value, both using it around 75% of the time (Figure 8). In Chrome, developers

TABLE IX
NUMBER OF VALUE COLLECTIONS AND UNIQUE EXPRESSIONS VIEWED

Bug	Tool	# collection actions		Unique expr viewed	
Bug1	Chrome	9 (9-12)	$p = 1.0$	7 (7-11)	$p = 0.91$
	Replay	11 (8-14)		8 (7-9)	
Bug2	Chrome	2 (1-2)	$p = 0.83$	2 (1-2)	$p = 0.83$
	Replay	8 (0-11)		6 (0-7)	
Bug3	Chrome	7 (6-22)	$p = 0.35$	4 (3-9)	$p = 0.21$
	Replay	14 (12-28)		9 (6-11)	
Bug4	Chrome	7 (0-13)	$p = 0.21$	6 (0-9)	$p = 0.17$
	Replay	16 (11-22)		10 (6-12)	

Note: numbers are reported as median (interquartile range)

need to first add a breakpoint and then rerun the program. When the execution is paused, they can hover over an identifier and the runtime value of that identifier will appear in a pop up. In Replay, developers first ensure the recording is rewound to when the variable is in scope, and then they can hover over the identifier to see its value. However, the hover pop up can only view expressions that appear in the source code and that ends with an identifier or field access (e.g. `arr[3].field`). If it ends with a function call (e.g. `Array.from(bytes)`), they need other methods to view it.

One such method is the "console", which both Chrome and Replay developers used approximately 5% of the time. When the execution is paused at a statement in Chrome or the recording is rewound to a statement in Replay, developers can type an expression into the console, and its value will be evaluated in that context. Two Chrome users (P3 and P7) and two Replay users (P3 and P19) made use of the console. Another method to view more complex expressions is the print statement. In Chrome, developers inserted print statements by modifying the code in the editor, rerunning the program, and then viewing the console output. In Replay, developers only needed to add the print statement in the recording, and Replay automatically updated the output in the console. The print statement constituted 7.4% of the runtime value collection actions for Chrome users and 18.4% for Replay users. Replay reduced the work involved in using the print statement, and we observed that developers used print statements more with Replay (Mann-Whitney U test $p = 0.0003$).

We analyzed the types of expressions that developers examined. We classified expressions into complex and simple categories. A "simple" expression is one that can be viewed using pop up, meaning that it ends with an identifier or field access. A "complex" expression is one that cannot be viewed using a pop up. We found that viewing complex expressions is relatively uncommon in both Chrome and Replay sessions. Only 3 of 20 Chrome sessions and 5 of 20 Replay sessions involved complex expression views. Additionally, 4.65% of all Chrome runtime value collections and 3.37% of Replay runtime value collections focused on complex expressions.

While developers with Replay used print statements more, they encountered some difficulties. To add a print statement in

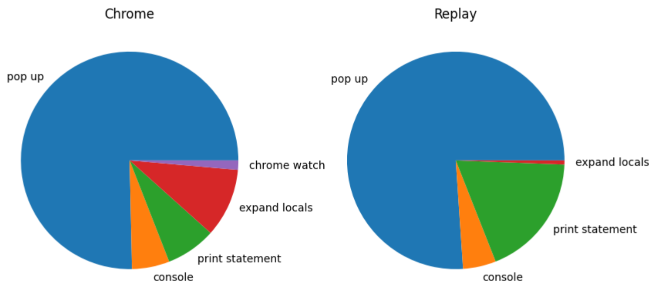
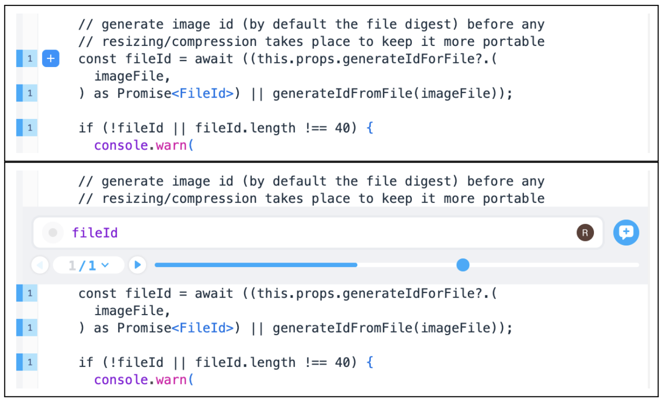


Fig. 8. The techniques developers used to collect runtime values


```
// generate image id (by default the file digest) before any
// resizing/compression takes place to keep it more portable
const fileId = await ((this.props.generateIdForFile?.(
  imageFile,
) as Promise<FileId>) || generateIdFromFile(imageFile));

if (!fileId || fileId.length !== 40) {
  console.warn(
```
```
// generate image id (by default the file digest) before any
// resizing/compression takes place to keep it more portable
fileId
1/1
const fileId = await ((this.props.generateIdForFile?.(
  imageFile,
) as Promise<FileId>) || generateIdFromFile(imageFile));

if (!fileId || fileId.length !== 40) {
  console.warn(
```

Fig. 9. The process of creating a print statement in Replay

Replay, the developer needs to click the plus sign (+), then a print statement panel will appear above the line (Figure 9). This is equivalent to adding a `console.log` statement above the line where the plus sign is clicked. When four participants (P4, P5, P8, P15) wanted to see the runtime value of a variable right after it was defined, they added the print statement by clicking on the definition line. Because the `console.log` statement was added above the definition line, this would result in a reference error. This was confusing for participants, and they sometimes attributed the reference error to an incorrect cause, such as that the statement was not executed.

In summary, we found that Chrome and Replay users collected similar numbers of runtime values, with most being collected in both cases using the hover pop up. Replay users used print statements more than Chrome users. Almost all runtime values collected are simple expressions.

VI. THREATS TO VALIDITY

Internal Validity. The results could be influenced by the order in which participants see the tasks, as familiarity gained from the first task might make the second easier. We mitigated the effect of ordering by alternating the orders of the tool and the task, creating eight experimental conditions in total.

External Validity. The generalizability of our findings may be limited by sampling bias, as our participants were primarily master's students and junior engineers. However, our demographic survey indicates that they had professional experience in software engineering. All participants had at least 7 years of programming experience, and 75% had at least 2 years of professional software engineering experience. Another potential threat to external validity is whether the selected tasks are representative of common debugging scenarios. To address this, we designed the study using a popular open-source web application and selected two bugs from its issue tracker. We did not choose the other two bugs from the issue tracker because we also wanted to include bugs that may happen during development, not only ones reported later by users. The four bugs in our study covered a range of debugging challenges, varying the size of the cause-effect chasms and the type of bug symptoms.

As with all lab studies, our participants worked with an unfamiliar codebase, which differs from real-world scenarios when engineers debug systems they are familiar with. To compensate for this, we provided participants with key knowledge needed to begin work on the tasks. We opted not to conduct a longitudinal study where participants could work on their own codebase, as we wanted to control the conditions to observe the causal impact of omniscient debuggers.

Our participants were first-time Replay users, and the results might differ for more experienced users who could potentially benefit more from the tool. While expert users would likely navigate the interface more fluently and encounter fewer usability issues, they would still face the same fundamental navigational limitations inherent to Replay's design, which we explore in the discussion section.

Construct Validity. The definition and interpretation of the codes in our code book may be subjective. To mitigate this threat, we built the initial code book from our video data, refined the definitions until two authors could independently and consistently annotate segments of the data, and demonstrated substantial to almost perfect agreement in inter-rater reliability.

VII. Discussion

In this paper, we conducted the first controlled experiment examining the impact of a commercial omniscient debugger, which was publicly released and sold as Replay. We found that, despite hopes that omniscient debuggers might dramatically ease debugging, they did not improve task success or completion time. This is similar to some prior evaluations of omniscient debuggers such as Timelapse [34] and Traceglasses [42], and in contrast to others [5], [37].

To examine why, we conducted the most detailed analysis to date of how omniscient debuggers change developer behavior. We found that participants faced two types of barriers: usability issues and fundamental navigational limitations. The usability issues included difficulties rewinding the execution trace to specific lines of code and confusing reference errors when print statements were not placed correctly. However, the more significant barriers were the limitations in how Replay supports navigation during debugging. We found that, despite expectations that stepping backward is the key feature and enabler of omniscient debuggers, providing better connection from symptom to cause may be more important. Replay has a "jump to code" feature that links events to the relevant code, but it was not particularly helpful. The ability to link the output to the corresponding code may be more important. Previous research prototypes have supported this, such as jumping to console [3] or graphical [25] output statements, or jumping to code that sets values affecting graphical elements [5]. Replay itself supports jumping to console output, but not graphical output. Since our tasks produced no console logs, participants only used jump to event.

It is important to distinguish between jump to event and to output. With jump to event, developers typically select an event that triggers the bug and then step forward. During the process, they may encounter irrelevant functions that they do not know if they need to step into. In Bugs 2 and 3, although there are only 2 function calls between the jump target and MS1 function, there are 5 other irrelevant functions that they need to decide whether to step into or not. After developers used the "jump to code" feature and found a starting point, their debugging experience was close to traditional breakpoint debugging, with limited support on where to step or what to inspect next. In contrast, when using jump to output, developers will typically choose the undesired output caused by the bug, which is after the bug, and then reason back. The backward reasoning approach is more aligned with omniscient debuggers, which allow developers to step backward, and which may have a much smaller branching factor, reducing the search space. The debugger can take advantage of this, with backward reasoning techniques like dynamic slicing. In Bugs 3 and 4, the error message comes from an errorMessage field in the app state. If Replay supported jump to output, the developer could first jump to the state update statement (the "Set state" box in Figure 5), and then trace back to find the statement that generated the error message. If the chasm between the state update and the error message was too large to step through, as in Bug 3 (5 function calls), the debugger could further help the developer by tracing back to the error generating code (the "Error" box) using slicing techniques. Tools might also add support for jumping from an HTML element to the React code that renders it, as we have noticed two participants (P11, P19) trying to locate the code manually.

We also found that some effective debugging strategies were underutilized, contrary to our expectations. For defects with an error message dialog, we expected developers to start debugging by locating the relevant code using the message (by searching for it, or clicking the link to the error in the console). However, only 8 out of 20 developers did so initially, and they reached MS1 much faster (median 1.8 min vs. 14 min; $p = 0.0033$). Although this strategy was effective, many developers did not use it, possibly because they were not familiar with it. This suggests that there are potential benefits to increasing awareness of different debugging strategies for various types of defects [47].

While the wide release of omniscient debuggers commercially is an important milestone in their development, our work suggests more remains to be done to ensure their value in practice. In particular, the more high-level interactions with the execution trace that research prototypes have explored is crucial to fully realizing the value of omniscient debugging.

Acknowledgments

We thank our study participants for their time. This research was funded in part by the National Science Foundation Grant CCF-1845508 and Accenture.

References

[1] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *International Conference on Software Engineering*, May 2018, pp. 572–583.

[2] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, Apr. 1997.

[3] B. Lewis, "Debugging Backwards in Time," in *International Workshop on Automated Debugging*, Sep. 2003.

[4] G. Pothier and É. Tanter, "Back to the future: Omniscient debugging," *Software*, vol. 26, no. 6, pp. 78–85, 2009.

[5] A. J. Ko and B. A. Myers, "Finding causes of program output with the Java Whyline," in *Conference on Human Factors in Computing Systems*, Apr. 2009, pp. 1569–1578.

[6] Undo, "Udb," 2025. [Online]. Available: https://undo.io/products/udb/

[7] Record Replay, Inc., "Replay," 2025. [Online]. Available: https://www.replay.io/

[8] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, Jul. 1982.

[9] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How Programmers Debug, Revisited: An Information Foraging Theory Perspective," *Transactions on Software Engineering*, vol. 39, no. 2, Feb. 2013.

[10] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Conference on Human Factors in Computing Systems*, 2013, p. 3063–3072.

[11] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *International Conference on Software Engineering*, 2005, p. 126–135.

[12] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983.

[13] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," in *Conference on Human Factors in Computing Systems*, 1986, p. 171–174.

[14] A. Alaboudi and T. D. LaToza, "Using hypotheses as a debugging aid," in *Symposium on Visual Languages and Human-Centric Computing*, 2020, pp. 1–9.

[15] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Conference on Human Factors in Computing Systems*, 2004, p. 151–158.

[16] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[17] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[18] M. Monperrus, "Automatic software repair: A bibliography," *Computing Surveys*, vol. 51, no. 1, Jan. 2018.

[19] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *Software*, vol. 23, no. 4, pp. 76–83, 2006.

[20] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *International Conference on Software Engineering*, 2018.

[21] A. Afzal and C. Le Goues, "A study on the use of ide features for debugging," in *International Conference on Mining Software Repositories*, 2018, p. 114–117.

[22] H. Eladawy, C. Le Goues, and Y. Brun, "Automated program repair, what is it good for? not absolutely nothing!" in *International Conference on Software Engineering*, 2024.

[23] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *International Symposium on Software Testing and Analysis*, 2011, p. 199–209.

[24] F. N. Meem, J. Smith, and B. Johnson, "Exploring experiences with automated program repair in practice," in *International Conference on Software Engineering*, 2024.

[25] H. Lieberman and C. Fry, "Bridging the gulf between code and behavior in programming," in *Conference on Human Factors in Computing Systems*, 1995.

[26] P. Gestwicki and B. Jayaraman, "Methodology and architecture of jive," in *Symposium on Software Visualization*, 2005, p. 95–104.

[27] J. Vilk, E. Berger, J. Mickens, and M. Marron, "Mcfly: Time-travel debugging for the web," *arXiv preprint arXiv:1810.11865*, Oct. 2018.

[28] A. Deiner and G. Fraser, "Nuzzlebug: Debugging block-based programs in scratch," in *International Conference on Software Engineering*, 2024.

[29] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Conference on Programming Language Design and Implementation*, 1990, pp. 246–256.

[30] A. Treffer and M. Uflacker, "The Slice Navigator: Focused Debugging with Interactive Dynamic Slicing," in *International Symposium on Software Reliability Engineering Workshops*, Oct. 2016.

[31] C. Thiede, M. Taeumel, and R. Hirschfeld, "Object-centric time-travel debugging: Exploring traces of objects," in *International Conference on the Art, Science, and Engineering of Programming*, 2023, p. 54–60.

[32] ——, "Time-awareness in object exploration tools: Toward in situ omniscient debugging," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2023, p. 89–102.

[33] A. Lienhard, J. Fierz, and O. Nierstrasz, "Flow-centric, back-in-time debugging," in *Objects, Components, Models and Patterns*, 2009, pp. 272–288.

[34] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Symposium on User Interface Software and Technology*, Oct. 2013, pp. 473–484.

[35] K. Y. Phang, J. S. Foster, and M. Hicks, "Expositor: Scriptable time-travel debugging with first-class traces," in *International Conference on Software Engineering*, 2013, pp. 352–361.

[36] M. Willembrinck, S. Costiou, A. Etien, and S. Ducasse, "Time-Traveling Debugging Queries: Faster Program Exploration," in *International Conference on Software Quality, Reliability and Security*, Dec. 2021.

[37] A. Alaboudi and T. D. Latoza, "Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses," in *Symposium on User Interface Software and Technology*, Oct. 2023, pp. 1–14.

[38] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *European Conference on Object-Oriented Programming*, 2008, pp. 592–615.

[39] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Conference on Object-Oriented Programming Systems, Languages and Applications*, 2007, p. 535–552.

[40] E. T. Barr and M. Marron, "Tardis: affordable time-travel debugging in managed runtimes," in *International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, p. 67–82.

[41] G. Pothier and É. Tanter, "Summarized trace indexing and querying for scalable back-in-time debugging," in *European Conference on Object-Oriented Programming*, 2011, pp. 558–582.

[42] K. Sakurai and H. Masuhara, "The omission finder for debugging what-should-have-happened bugs in object-oriented programs," in *Symposium on Applied Computing*, 2015, p. 1962–1969.

[43] K. Sakurai, "Traceglasses: A trace-based debugger for realizing efficient navigation," *IPSJ Transaction on Programming*, vol. 3, no. 3, p. 1, 2010.

[44] R. Wang and T. D. LaToza, "Supplemental material of How Omniscient Debuggers Impact Debugging Behavior," 8 2025. [Online]. Available: https://figshare.com/articles/dataset/Supplemental_material_of_How_Omniscient_Debuggers_Impact_Debugging_Behavior/29852750

[45] Excalidraw contributors, "Excalidraw," 2024. [Online]. Available: https://github.com/excalidraw/excalidraw

[46] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, 1977.

[47] T. D. LaToza, M. Arab, D. Loksa, and A. J. Ko, "Explicit programming strategies," *Empirical Software Engineering*, vol. 25, no. 4, p. 2416–2449, Mar. 2020.