Interactive Record/Replay for Web Application Debugging

Brian Burg[1], **Richard Bailey**[1], **Andrew J. Ko**[2], **Michael D. Ernst**[1]

Computer Science and Engineering[1]
University of Washington
{burg, rjacob, mernst}@cs.washington.edu

The Information School[2]
University of Washington
ajko@uw.edu

ABSTRACT

During debugging, a developer must repeatedly and manually reproduce faulty behaviors in order to inspect different facets of the program's execution. Existing tools for reproducing such behaviors prevent the use of debugging aids such as breakpoints and logging, and are not designed for interactive, random-access exploration of recorded behavior. This paper presents Timelapse, a tool for quickly recording, reproducing, and debugging interactive behaviors in web applications. Developers can use Timelapse to browse, visualize, and seek within recorded program executions while simultaneously using familiar debugging tools such as breakpoints and logging. Testers and end-users can use Timelapse to demonstrate failures *in situ* and share recorded behaviors with developers, improving bug report quality by obviating the need for detailed reproduction steps. Timelapse is built on Dolos, a novel record/replay infrastructure that ensures deterministic execution by capturing and reusing program inputs both from the user and from external sources such as the network. Dolos introduces negligible overhead and does not interfere with breakpoints and logging. In a small user evaluation, participants used Timelapse to accelerate existing reproduction activities, but were not significantly faster or more successful in completing the larger tasks at hand. Together, the Dolos infrastructure and Timelapse developer tool support systematic bug reporting and debugging practices.

Author Keywords

Debugging, deterministic replay, web applications

ACM Classification Keywords

D.2.5 Software Engineering: Testing and debugging

General Terms

Human Factors; Design

INTRODUCTION

Debugging is often an iterative process in which developers repeatedly adjust their view on a program's execution by adding and removing breakpoints and inspecting program state at different times and locations. This iteration requires a developer to repeatedly reproduce the behavior they are inspecting, which can be time-consuming and error-prone [36]. In the case of interactive programs, even reproducing a failure can be difficult or impossible: failures can occur on mouse drag events, be time-dependent, or simply occur too infrequently for a developer to easily reach a program state suitable for debugging.

Deterministic record/replay [5] is a technique that can be used to record a behavior once and then deterministically replay it repeatedly and automatically, without user interaction. Though it seems that the capability to record and replay executions should be useful for debugging, no prior work has described actual use cases for these capabilities, or how to best expose these capabilities via user interface affordances. At most, prior systems demonstrate feasibility by providing a simple VCR-like interface [18, 33] that can record and replay linearly. Many record/replay systems have no UI, and are controlled via commands to the debugger or special APIs [9, 11, 27]. Finally, prior work does not consider how record/replay capabilities can interoperate with and enhance existing debugging tools like breakpoints and logging. Prior tool instantiations [18, 25, 1] are inappropriate for debugging because they can interfere with program performance, determinism, and breakpoint use, and they are difficult to deploy.

This paper presents Timelapse, a developer tool for capturing and replaying web application behaviors during debugging, and Dolos, a novel record/replay infrastructure for web applications. A developer can use Timelapse's interactive visualizations of program inputs to find and seek non-linearly through the recording, and then use the debugger or other tools to understand program behavior. To ensure deterministic execution, Dolos captures and reuses user input, network responses, and other nondeterministic inputs as the program executes. It does this in a purely additive way—without impeding the use of other tools such as debuggers—via a novel adaptation of virtual machine record/replay techniques to the execution environment of web browsers. Debugging tools are particularly important for interactive web applications because the web's highly dynamic, event-driven programming model stymies traditional static program analysis techniques.

This paper makes the following contributions:

- Dolos: a fast, scalable, precise, and practical infrastructure for deterministic record/replay of web applications.
- Timelapse: a developer tool for creating, visualizing, and navigating program recordings during debugging tasks.
- The first user study to explore how and when developers use record/replay tools during realistic debugging tasks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UIST 2013, October 8–11, 2013, St Andrews, United Kingdom.
Copyright 2013 ACM 978-1-4503-2268-3/13/10...$15.00.

Dolos and Timelapse are free software: our source code, study materials, and data are available on the project website[1].

We first discuss the design of Timelapse, and use a scenario to illustrate how Timelapse supports recording, reproducing, and debugging interactive behaviors. Next, we present the design and implementation of the underlying Dolos record/replay infrastructure. We discuss the results of a small study to see how developers use Timelapse in open-ended debugging tasks. Finally, we conclude with related work and present several implications for future interactive debugging tools.

REPRODUCING AND NAVIGATING PROGRAM STATES

The Timelapse developer tool is designed around two activities: capturing user-demonstrated program behaviors, and quickly and repeatedly reaching program states within a recording when localizing and understanding a fault. The novel features we describe in this section support these activities by making it simple to record program behavior and by providing visualizations and affordances that quicken the process of finding and seeking to relevant program states without manually reproducing behavior.

To better understand the utility of replay capabilities during debugging, we first conducted a small pilot study with a prototype record/replay interface. Using contextual inquiry, we found that developers primarily used the prototype to isolate buggy output, and to quickly reach specific states when working backwards from buggy output towards its root cause. Towards these ends, we saw several common use cases: “play and watch”; isolating output using random-access seeking; stepping through execution in single-input increments, and reading low-level input details or logged output.

This section introduces the novel features of the Timelapse developer tool by showing how a fictional developer named Claire might use Timelapse’s features while debugging.

Debugging Scenario: (Buggy) Space Invaders

Claire, a new hire at a game company, has been asked to fix a bug in a web application version of the Space Invaders video game[2]. In this game, the player moves a defending ship and shoots advancing aliens. The game’s implementation is representative of modern object-oriented interactive web programs: it uses timer callbacks, event-driven programming, and helper libraries. The game contains a defect that allows multiple player bullets to be in flight at a time; there is only supposed to be one player bullet at a time (Figure 1).

Reproducing Program Behavior

Claire is unfamiliar with the Space Invaders implementation, so her first step towards understanding and fixing the multiple-bullet defect is to figure out how to reliably reproduce it. This is difficult because the failure only occurs in specific game states and is influenced by execution conditions outside of her control, such as random numbers, the current time, or asynchronous network requests.

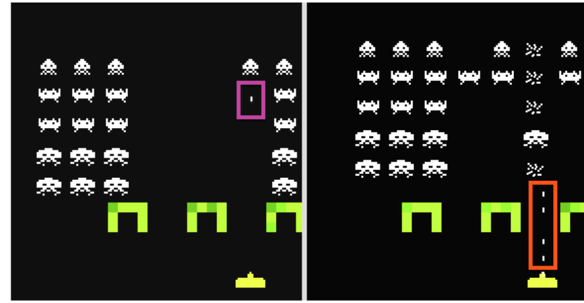

Figure 1. Screenshots of normal and buggy Space Invader game mechanics. Only one bullet should be in play at a time (shown on left). Due to misuse of library code, each bullet fires two asynchronous `bulletDied` events instead of one event when it is destroyed. The double dispatch sometimes enables multiple player bullets being in play at once (shown on right). This happens when two bullets are created: one between two `bulletDied` events, and the other after both events.

With Timelapse, Claire begins capturing program behaviors with a single click (Figure 2-6), plays the game until she reproduces the failure, and then finishes the recording. Recordings created by Timelapse are compact, self-contained, and serializable, so Claire can attach her recording to a bug report or share it via email.

To reproduce the defect with traditional tools, Claire would have to multitask between synthesizing reproduction steps, playing the video game, and reflecting on whether the reproduction steps are correct. Once Claire finds reliable reproduction steps, she could then use breakpoints to further understand

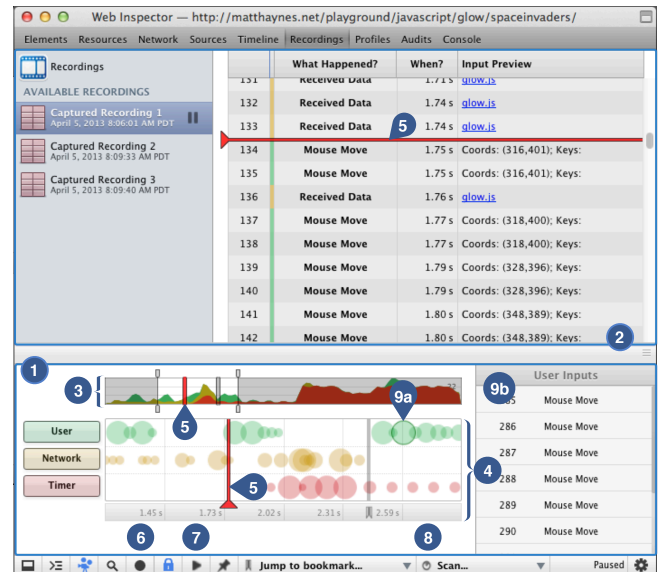

Figure 2. The Timelapse tool interface presents multiple linked views of recorded program inputs. Above, the timelines drawer (1) is juxtaposed with a detailed view of program inputs (2). The recording overview (3) shows inputs over time with a stacked line graph, colored by category. The overview’s selected region is displayed in greater detail in the heatmap view (4). Circle sizes indicate input density per category. In each view, the red cursor (5) indicates the current replay position and can be dragged. Buttons allow for recording (6), 1× speed replay (7), and breakpoint scanning (8). Details for the selected circle (9a) are shown in a side panel (9b).

[1] `http://github.com/burg/timelapse/`

[2] `http://matthaynes.net/playground/javascript/glow/spaceinvaders/`

the defect. But, breakpoints might themselves affect timing, making the defect harder to trigger or requiring modified reproduction steps.

Navigating to Specific Program States

To focus her attention on code relevant to the failure, Claire needs to know which specific user input—and thus which event handler invocations—caused the second bullet to appear.

Claire uses Timelapse's visualization and navigation tools (Figure 2) to locate and seek the recording to an instance of the multiple bullets failure. First, Claire limits the zoom interval to when she actually fired bullets, and then filters out non-keystroke inputs. She replays single keystrokes with a keyboard shortcut until a second bullet is added to the game board. Then, she seeks execution backward by one keystroke. At this point, she is confident that the code which created the second bullet ran in response to the current keystroke.

Without Timelapse, it would not be possible for Claire to isolate the failure to a specific keystroke and then work backwards from the keystroke. Instead, she would have to insert logging statements, repeatedly reproduce the failure to generate logging output, and scrutinize logged values for clues leading towards the root cause.

Navigation Aid: Debugger Bookmarks

Having tracked down the second bullet to a specific user input, Claire now needs to investigate what code ran, and why.

With Timelapse, Claire sets several *debugger bookmarks* (Figure 3-6) at positions in the recording that she wants to quickly revert back to, such as the keystroke that caused the second bullet to appear or an important program point reached via the debugger. Debugger bookmarks support the concept of temporal focus points [28],which are useful when a developer wants to relate information [13]—such as the program's state at a breakpoint hit—that is only available at certain points of execution. Timelapse restores a debugger bookmark by seeking to the preceding input, setting and continuing to the preceding breakpoint, and finally simulating the user's sequence of debugger commands (step forward/into/out).

With traditional tools, Claire must explore an execution with debugger commands such as "step into", "step over", and "step out". This is frustrating because these commands are irreversible, and Claire would have to manually reproduce the failure multiple times to compare multiple instants or the effects of different commands.

Navigation Aid: Breakpoint Radar

Once Claire finds the code that creates bullets, she needs to understand why some keystrokes fire bullets and others do not.

With Timelapse, Claire first adjusts the zoom interval to include keystrokes that did and did not trigger the failure. Then, she sets a breakpoint inside the `Bullet.create()` method and records and visualizes when it is actually hit during the execution using the *breakpoint radar* feature (Figure 3-3a). Breakpoint radar automates the process of replaying the execution, pausing and resuming at each hit, and visualizing when

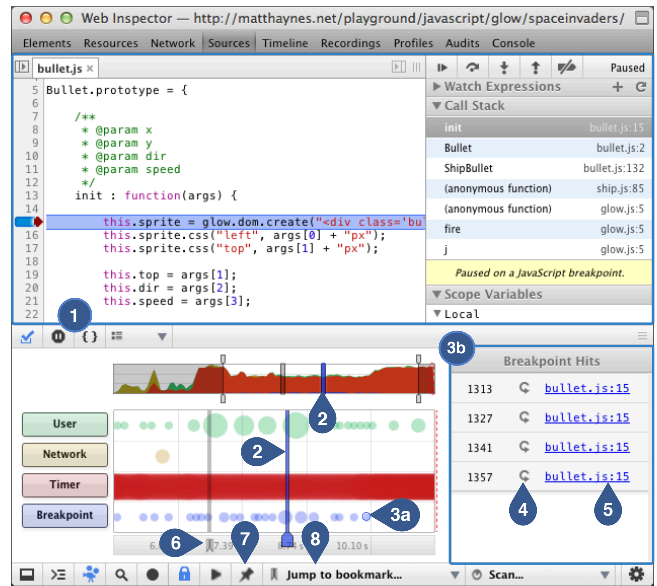

Figure 3. Timelapse's visualization of debugger status and breakpoint history, juxtaposed with the existing Sources panel and debugger (1). A blue cursor (2) indicates that replay execution is paused at a breakpoint, instead of between user inputs (as shown in Figure 2). Blue circles mark the location of known breakpoint hits, and are added or removed automatically as breakpoints change. A side panel (3b) shows the selected (3a) circle's breakpoints. Shortcuts allow for jumping to a specific breakpoint hit (4) or source location (5). Debugger bookmarks (6) are set with a button (7) and replayed to by clicking (6) or by using a drop-down menu (8).

the debugger paused. Claire can easily see which keystrokes created bullets and which did not.

With traditional tools, Claire would need to repeatedly set and unset breakpoints in order to determine which keystrokes did or *did not* create bullets. To populate the breakpoint radar timeline, Claire would have to manually hit and continue from dozens or hundreds of breakpoints, and collect and visualize breakpoint hits herself. For this particular defect, breakpoints interfere with the timing of the bullet's frame-based animations, so it would be nearly impossible for Claire to pause execution when two bullets are in flight.

Interacting with Other Debugging Tools

Once Claire localizes the part of the program responsible for the multiple bullets, she still needs to isolate and fix the root cause. To do so, Claire uses debugging strategies that do not require Timelapse, but nonetheless benefit from it. Timelapse is designed to be used alongside other debugging tools such as breakpoints[3], logging, and element inspectors; its interface can be juxtaposed (Figure 3) with other tools.

Through code inspection, Claire observes that the creation of a bullet is guarded by a flag indicating whether any bullets are already on the game board. The flag is set inside the `Bullet.create()` method and cleared inside the `Bullet.die()` method. To test her intuition about the

[3]To prevent breakpoints from interfering with tool use cases, Timelapse tweaks breakpoints in several ways: breakpoints are disabled when recording or seeking, and enabled during real-time playback.

code's behavior, she inserts logging code and captures a new execution to see if the method calls are balanced. The logging output in Figure 4 is synchronized with the replay position: as Claire seeks execution forward or backward, logging output up to the current instant is displayed. Logging output is cleared when a fresh execution begins (i.e., Timelapse seeks backwards) and then populated as the program executes normally.

Claire has discovered that the multiple-bullet defect is caused by the `bulletDied` event being fired twice, allowing a second replacement bullet to be created if the bullet "fire" key is pressed between the two event dispatches. In other words, the failure is triggered by firing a bullet while another bullet is being destroyed (by collision or leaving the game board).

With basic record/replay functionality, affordances for navigating the stream of recorded inputs, and the ability to easily reach program states by jumping directly to breakpoint hits, Timelapse both eliminates the need for Claire to repeatedly reproduce the Space Invaders failure and frees her to focus on understanding the program's logic. Our user evaluation explores these benefits further.

RECORD/REPLAY INFRASTRUCTURE

Dolos is the underlying record/replay infrastructure that enables the Timelapse tools and user interfaces. For Timelapse to be useful during debugging tasks, Dolos must not disrupt existing developer tools and workflows. Concretely, this entails the following four design goals:

1. **Low overhead.** Recording must introduce minimal performance overhead, because many interactive web applications are performance-sensitive. Replaying must be fast so that users can quickly navigate through recordings.

2. **Deterministic replay.** Recordings must exactly reproduce observable program behavior when replaying. Replaying should not have external effects.

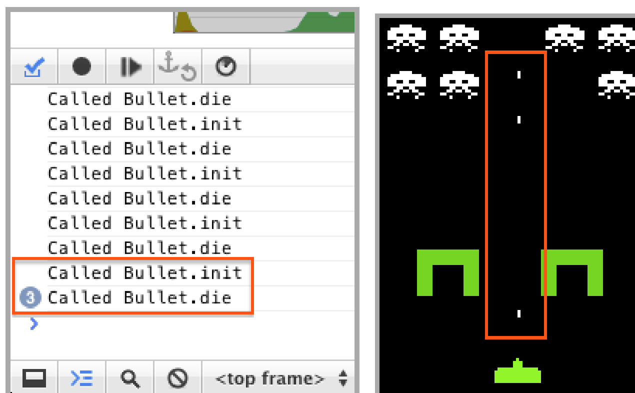

Figure 4. Screenshots of the logging output window and the defect's manifestation in the game. Claire added logging statements to the `die` and `init` methods. At left, the logging output shows the order of method entries, with the blue circle summarizing 3 identical logging outputs. According to the last 4 logging statements (outlined in red), calls to `die` and `init` are unbalanced. At right, three in-flight bullets correspond to the three calls to `Bullet.init`.

3. **Non-interference.** Recording and replaying must not interfere with the use of tools such as breakpoints, watchpoints, profilers, element inspectors, and logging.

4. **Deployability.** Recording and replaying must require no special installation, configuration, or elevated user privileges.

To the best of our knowledge, no prior record/replay tool satisfies all of these constraints. Source-level instrumentation of JavaScript [16] can perturb performance [25] and relies on hard-to-deploy[4] proxies to perform source rewriting. Rewriting techniques can interfere with a developer's expectations by obfuscating source code or by causing the debugger to unexpectedly interact with code the developer did not write. User-space record/replay library tools [18, 27] execute in the same address space as the target program and use wrappers to interpose on nondeterministic APIs (such as the `Date` constructor). These tools are inherently incompatible with breakpoint debuggers: because they piggyback on the target program, pausing at a breakpoint will also prevent the record/replay library's mechanisms from executing. Virtual machine replay debugging [33] ensures deterministic execution of the entire browser instead of one page context—preventing the use of built-in debugging tools which cannot run independently from other browser components. Macro-like replay tools such as Selenium WebDriver [31], CoScripter [15], or Sikuli [34] are commonly used for workflow automation; towards that end, they only record and replay user input, and not other program inputs such as network traffic, dates and times, and other sources of non-determinism. Replaying only user input is not deterministic enough to consistently reproduce web application behaviors. These external tools are unaware of internal system state or external server state: for example, they may try to provide input while the program is paused at a breakpoint, or cause the program to ask a server for resources that are no longer valid.

Web Browser Architecture

We use the term *web interpreter* to refer to the interpreter-like core of web browsers. The web interpreter processes inputs from the network, user, and timers; executes JavaScript; computes page layout; and renders output. We do not consider features that do not affect the results of computation, such as bookmarks and a browser's address bar. The web interpreter schedules asynchronous computation using a cooperative single-threaded event loop. It communicates with embedders (Safari, Google Chrome) and platforms (Linux, Mac OS X) via embedder and platform APIs (Figure 5). Both APIs are bidirectional: the web interpreter can call "out" to collect environmental information from the platfrom or delegate security policies to the embedder; the platform and embedder can call "in" to schedule computation (asynchronous timers, user input, network traffic) in the web interpreter's event loop.

[4] Proxies used for instrumentation and debugging like Fiddler [29] require elevated privileges, manual configuration, and intentional man-in-the-middle attacks to subvert SSL encryption.

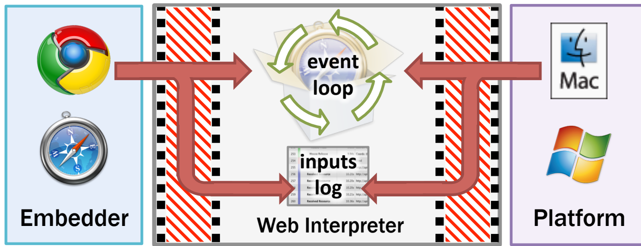

Figure 5. The flow of inputs while recording execution. Boxes delineate module boundaries. Arrows indicate the flow of program inputs. The web interpreter receives input and sends output via embedder and platform APIs (thick lines). During recording, input shims (striped regions) record program inputs delivered via the public APIs, and pass on the inputs to the unmodified event loop.

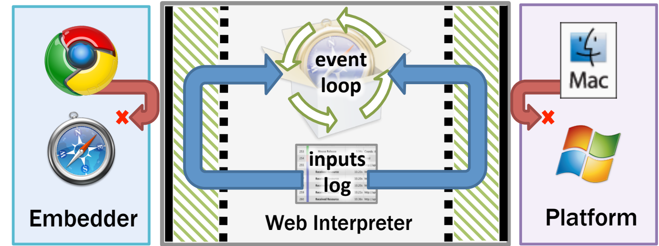

Figure 6. The flow of inputs while replaying execution. During replay, the interpreter's replay controller uses the input log to simulate the sequence of recorded public API calls into the interpreter (blue arrows). External calls to public APIs are ignored (solid black line) to prevent user interactions from causing a different execution. Calls from the web interpreter to platform APIs (i.e., to get the current time) use memoized return values and do not have external effects.

Design

Our requirements of low overhead, deterministic replay, non-interference, and deployability are closest to the design goals of virtual machine-based record/replay systems. Well-known systems in this space, such as ReVirt [7] and VMWare Replay Debugging [33], achieve deterministic record/replay via the "instruction-counting trick". A hypervisor interposes on nondeterministic machine instructions, but executes ordinary instructions natively on the bare-metal processor. Interrupts are captured and injected accurately by counting their position in the instruction stream. This technique satisfies our four design goals: it is fast, ensures precise replay, is compatible with debuggers, and requires minimal modifications to the runtime environment.

The execution model of web programs does not have machine instructions or interrupts, but there are parallels between virtual machines and web browsers. The Dolos infrastructure makes the following novel substitutions to make the "instruction-counting trick" work for the web's execution model:

- Instead of simulating deterministic hardware, Dolos simulates deterministic responses from embedder and platform when the web interpreter calls out to them.
- Instead of hardware interrupts, Dolos captures and simulates the subset of inbound embedder and platform API calls that trigger computation within the web application.
- Instead of counting instructions as a unit of execution progress, Dolos counts DOM event dispatches—precursors to the execution of JavaScript code—that may have deterministic or nondeterministic effects.

Implementation

The Dolos infrastructure is a modified version of the WebKit browser toolkit. We chose WebKit because it contains the most widely-deployed web interpreter (WebCore) and developer toolchain (Web Inspector). Our deployment model is straightforward: a version of WebKit built with Timelapse and Dolos support is distributed as a dynamic library, and can be used with existing WebKit embedders such as Safari or Google Chrome by adjusting the dynamic library load path.

Recording and replaying

Hypervisors (and record/replay systems based on them) are fast in part because most execution occurs at full speed on bare hardware. Similarly, Dolos achieves high record/replay performance in part by using the same code paths as normal execution when recording and replaying program inputs. Dolos captures and replays calls to the embedder and platform APIs using *shims*—thin layers used to observe external API calls or simulate external API calls from within. During recording (Figure 5), the shims record all API calls that affect program determinism. During replay, these API calls are simulated (Figure 6) to deterministically reproduce the recorded program behavior. To prevent user interactions from causing a different execution during replay, the shims block external API calls while allowing simulated calls to proceed.

During recording, Dolos saves the timing, ordering, and details of inbound calls to the web interpreter that are pertinent to the recorded web program's execution. During replay, these inbound calls are simulated by Dolos, rather than actually being issued by the embedder or platform. Dolos's approach to capturing and replaying computation-triggering calls is applicable to programs implemented using an event-loop-based UI framework. Dolos can seek to and/or pause execution prior to any simulated inbound call. Because the event loop is transparent to web programs, Dolos "pauses" execution without using breakpoints by simply not simulating the next computation-triggering inbound call. One can use breakpoints to pause JavaScript execution at any statement, but cannot use them to pause execution before the web interpreter interprets an inbound call.

Program inputs

Like hypervisor record/replay systems, Dolos classifies inputs to the web interpreter—that is, relevant inbound and outbound embedder and platform API calls—as either interrupts or environmental nondeterminism. Table 1 enumerates the major sources of interrupt-like inputs and environmental inputs. Conceptually, interrupts cause computation to happen, while environmental inputs embody nondeterministic aspects of the environment. Interrupts are API calls *from* the embedder or platform to the interpreter; environmental inputs are API calls *to* the embedder or platform from the interpreter.

Input	Classification	DOM Derivatives
Keyboard strokes	Interrupt	keyboard events
Mouse input	Interrupt	mouse events
Timer callbacks	Interrupt	(none)
Asynchronous events	Interrupt	animation events
Network response	Interrupt	AJAX, images, data
Random values	Environment	Math.random
Current time	Environment	Date.now
Persistent state	Environment	cookies, local storage

Table 1. Major classes of input as defined by Dolos.

Dolos records and replays embedder/platform API calls rather than individual DOM events [18, 21, 25] which result from the web interpreter's interpretation of inbound API calls. For example, instead of recording individual DOM `click` events, Dolos captures the embedder's calls to the web interpreter's `handleMousePress` API. This in turn may trigger one, many, or no DOM event dispatches, depending on the interpreter's state. Dolos's strategy reduces log size, runtime overhead, and implementation complexity because it does not require event targets to be serialized when recording or re-targeted when replaying.[5]

Dolos captures and replays network traffic in the same way that other event loop callbacks are handled. When capturing, Dolos caches the HTTP headers and raw data of network responses as they are passed from the embedder to the web interpreter. When replaying, Dolos blocks outbound network requests from actually reaching the network. Dolos simulates deterministic network responses by reusing cached HTTP headers and data. For example, when loading images asynchronously on Flickr, Dolos caches all image data when capturing. When replaying, Dolos simulates network responses by reusing cached image data, and never communicates with Flickr servers.

For environmental inputs, Dolos uses shims to memoize the web interpreter's C++ calls to the platform or embedder APIs rather than memoizing calls to nondeterministic JavaScript functions. For example, Dolos does not record the return value of JavaScript's `Date.now()` function; instead, Dolos memoizes the JavaScript engine's calls to the `currentTimeMS()` platform API inside the implementation of `Date.now()`.

Replay fidelity

The Dolos infrastructure guarantees identical execution when recording and replaying, up to the order and contents of DOM events dispatched. JavaScript execution is completely deterministic. There is no guarantee regarding number of layout reflows or paints due to time compression or internal browser nondeterminism. This is unimportant because visual output cannot affect the determinism of JavaScript computation: JavaScript client code can ask for computed layout data, but will block until the layout algorithm runs on the current version of the DOM tree. Rendering and painting activity is not exposed to client code at all.

To the best of our knowledge, Dolos is the first web record/replay infrastructure that *detects* execution divergence. Timelapse warns users when divergences are detected or when known-nondeterministic APIs are used, and allows them to abort replay, ignore divergences, or report feature requests to the Dolos developers. Dolos uses differences in measures such as DOM node counts and event dispatch counts to detect unexpected execution divergences caused by bugs in Dolos itself. This has helped us find and address obscure sources of nondeterminism, such as resource caching effects, asynchrony in the HTML parser, implicit window size and focus state, and improper multiplexing of inputs to nested `iframe` elements.

The Dolos prototype does not address all known sources of nondeterminism, such as the Touch, Battery, Sensor, Screen, or Clipboard APIs, among others. There are no conceptual barriers to supporting these features: they are implemented in terms of standardized DOM events and interfaces, making them relatively easy to interpose upon using techniques described in this paper. Each new program input requires local changes to route control flow through a shim, plus a helper class to serialize and simulate the program input.

Engineering cost

Dolos's design scales to new platforms, embedders, and sources of nondeterminism because inputs are recorded and replayed at existing module boundaries. Implementing Dolos required minimal changes to WebKit's architecture: namely, the insertion of shims just beneath the web interpreter's public embedder and platform APIs (shown in Figure 5 and in Figure 6). The Dolos infrastructure consists of 7.6K SLOC (74 new files, 75 modified files). For comparison, WebKit contains about 1.38M SLOC.

Overhead

Dolos's record/replay performance slowdown is unnoticeable ($< 1.1\times$) for interactive workloads and modest ($\leq 1.65\times$) for non-interactive benchmarks without any significant optimization efforts (Table 2). Recording overhead and the amount of data collected scales with user, network, and nondeterministic inputs, not CPU time.

We report the geometric mean of 10 runs (except for interactive runs, which were recorded once but replayed 10 times). We cleared network resource caches between executions to avoid memory and disk cache nondeterminism. We used local copies of benchmarks to avoid network nondeterminism.

Recording has almost no time overhead: execution times are dominated by the subject program. Replaying at $1\times$ speed is marginally slower than a normal execution due to extra work performed by Dolos, and seeking (fast replaying) is much faster because it elides user and network waits from the recorded execution.

While being created or replayed, recordings are stored in-memory and consume modest amounts of memory (first column in the data size section of Table 2). When serialized, the recordings are highly compressible. A recording's length is limited only by main memory; in our experience, users attempt to minimize recording length to reduce the number of irrelevant inputs. Timelapse's linear presentation of time does not scale well to long executions. This could be ameliorated by using nonlinear time scaling techniques [32, 14].

[5] Guo et al. [9] report on the space and time benefits of memoizing application-level API calls instead of low-level system calls [27].

Program			Run Time						Data Size (KB)	
Name	Description	Workload	Bottleneck	Baseline	Disabled	Recording	Replaying	Seeking	Log	Site
JSLinux	x86 emulator	Run until login	network	10.5s	1.00×	1.65×	1.65×	0.37×	24.7 / 71.4 / 8.5	4500
JS Raytracer	ray-tracer	Complete run	CPU	6.3s	1.00×	1.01×	1.17×	1.02×	24.0 / 69.4 / 10.2	5.9
Space Invaders	video game	Scripted gameplay	timers	25.8s	1.00×	1.03×	1.22×	0.25×	247 / 683 / 56.8	712
Mozilla.org	home page	Read latest news	user	22.3s	1.00×	1.00×	1.09×	0.23×	187 / 502 / 29.5	2800
CodeMirror	text editor	Edit a document	user	16.6s	1.00×	1.00×	1.03×	0.07×	57.6 / 163 / 17.1	168
Colorpicker	jQuery widget	Reproduce defect	user	15.3s	1.00×	1.00×	1.07×	0.13×	112 / 302 / 26.7	577
DuckDuckGo	search engine	Browse results	user	14.1s	1.00×	1.00×	1.08×	0.19×	119 / 309 / 29.6	1900

Table 2. Overhead for three non-interactive and four interactive programs. "Baseline" is unmodified WebKit, and "Disabled" is Dolos when neither record nor replay is enabled. Log size is given for the in-memory representation, the uncompressed log file, and the compressed log file. Site content is images, scripts, and HTML.

Limitations

Dolos only ensures deterministic execution for *client-side* portions of web applications; it records and simulates client interactions with a remote server, but does not directly capture server-side state. Tools that link client- and server-side execution traces [35] may benefit from the additional runtime context provided by a Dolos recording. Dolos cannot control the determinism of local, external software components such as Flash, Silverlight, or other plugins. However, plugins interact via the embedder API; Dolos could handle plugin nondeterminism in the same way that other embedder nondeterminism is handled.

Web interpreters provide many API calls that do not affect program determinism. For example, WebKit's web interpreter includes APIs for usability features like native spell-checking, in-page search, and accessibility. Dolos does not record or replay these API calls because a developer may wish to use such features differently during replay, and these features do not affect the determinism of the web program.

Dolos's hypervisor-like record/replay strategy relies on the layered architecture of WebKit. It is not directly applicable to systems without clear API boundaries between the embedder, platform, and web interpreter. For example, the Gecko web interpreter used by Firefox is implemented by dozens of decoupled components rather than a monolithic module. This design makes it easy to extend the browser, but difficult to record and replay only the subset of components that affect the specific web program's execution, as opposed to those are used by browser features, browser extensions, or several web programs at once. Shims for Gecko would have the same behavior as Dolos's shims, but they would be placed throughout the web interpreter rather than at coarse abstraction boundaries.

HOW DO DEVELOPERS USE TIMELAPSE?

Prior work [7, 18, 27] asserts the usefulness of deterministic record and replay for debugging. In this section, we present a formative user study that investigates when, how, and for whom record/replay tools and specifically Timelapse are beneficial. Our specific research questions were:

RQ 1 How does Timelapse affect the way that developers reproduce behavior?

RQ 2 How do successful and unsuccessful developers use Timelapse differently?

Study Design

We recruited 14 web developers, 2 of which we used in pilot studies to refine the study design. Each participant performed two tasks. We used a within-subjects design to see how Timelapse changed the behavior of individual developers. For one task, participants could use the standard debugging tools included with the Safari web browser. For the other task, they could also use Timelapse. To mitigate learning effects, we randomized the ordering of the two tasks, and randomized task in which they were allowed to use Timelapse.

The goal of this study was to explore the variation in how developers used Timelapse, so the tasks needed to be challenging enough to expose a range of task success. To balance realism with replicability, we chose two tasks of medium difficulty, each with several intermediate milestones. Based on our results, our small exploratory study was still sufficiently large to capture the variability in debugging and programming skill among web developers.

Participants

We recruited 14 web developers in the Seattle area. Each participant had recently worked on a substantial interactive website or web application. One half of the participants were developers, designers, or testers. The other half were researchers who wrote web applications in the course of their research. We did not control for experience with the jQuery or Glow libraries used by the programs being debugged.

Programs and Tasks

Space Invaders. One program was the Space Invaders game from our earlier example scenario. The program consists of 625 SLOC in 6 files (excluding library code) and uses the Glow JavaScript library[6]. We chose this program for two reasons: its extensive use of timers makes it a heavy record/replay workload, and its event-oriented implementation is representative of object-oriented model-view programs, the dominant paradigm for large, interactive web applications.

We asked participants to fix two Space Invaders defects. The first was an API mismatch that occurred when we upgraded the Glow library to a recent version while preparing the program for use in our study. In prior versions, a sprite's coordinates were represented with `x` and `y` properties; in recent versions, coordinates are instead represented with `left` and `top` properties, respectively. After upgrading, the game's hit detection

[6] **http://www.bbc.co.uk/glow/**

code ceases to work because it references the obsolete property names. The second defect was described in the motivating example and was masked by the first defect.

Colorpicker. The other program was Colorpicker[7], an interactive widget for selecting colors in the RGB and HSV colorspaces (see in Figure 7). The program consists of about 500 LOC (excluding library and example code). The widget supports color selection via RGB (red, green, blue) or HSV (hue, saturation, brightness) component values or through several widgets that visualize dimensions of the HSV colorspace.

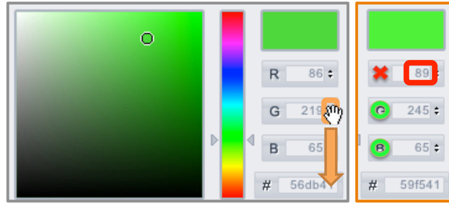

Figure 7. The Colorpicker widget.

We chose this program because it makes extensive use of the popular jQuery library, which—by virtue of being highly layered, abstracted, and optimized—makes reasoning about and following the code significantly more laborious.

The Colorpicker task was to create a regression test for a real, unreported defect in the Colorpicker widget. The defect manifests when selecting a color by adjusting an RGB component value, as shown in Figure 7. If the user drags the G component (left panel, orange), the R component spontaneously changes (right panel, red). The R component should not change when adjusting the G component. The bug is caused by unnecessary rounding in the algorithm that converts values between RGB and HSV color spaces. Since the color picker uses the HSV representation internally, repeated conversions between RGB and HSV can expose numerical instability during certain patterns of interaction.

We claim that both of these faults are representative of many bugs in interactive programs. Often there is nothing wrong with the user interface or event handling code *per se*, but faults that are buried deep within the application logic are only uncovered by user input or manifest as visual output. The Space Invaders faults lie in incorrect uses of library APIs, but manifest as broken gameplay mechanics. Similarly, the Colorpicker fault exists in a core numerical routine, but is only manifested ephemerally in response to mouse move events.

Procedure

Participants performed the study alone in a computer lab. Participants were first informed of the general purpose and structure of the study, but not of our research questions to avoid observer and subject expectancy effects. Immediately prior to the task where Timelapse was available, participants spent 30 minutes reading a Timelapse tutorial and performing exercises on a demo program. In order to proceed, participants were required to demonstrate mastery of recording, replaying, zooming, seeking, and using breakpoint radar and debugger bookmarks. Participants could refer back to the tutorial during subsequent tasks.

[7] `http://www.eyecon.ro/colorpicker/`

Each task was described in the form of a bug report that included a brief description of the bug and steps to reproduce the fault. At the start of each task, the participant was instructed to read the entire bug report and then reproduce the fault. Each task was considered complete when the participant demonstrated their correct solution. Participants had up to 45 minutes to complete each task. We stopped participants when they had demonstrated successful completion to us or exceeded the time limit.

After both task periods were over, we interviewed participants for 10 minutes about how they used the tool during the tutorial and tool-equipped task and how they might have used the tool on the other task. We also asked about their prior experience in bug reproduction, debugging, and testing activities.

Data Collection and Analysis

We captured a screen and audio recording of each participant's session, and gathered timing and occurrence data by reviewing the video recordings after the study concluded.

Our tasks were both realistic and difficult so as to draw out variations in debugging skill and avoid imposing a performance ceiling. We measured task success via completion of several intermediate task steps or critical events. For the Space Invaders task, the steps were: successful fault reproduction, identifying the API mismatch, fixing the API bug, reproducing the rate-of-fire defect, written root cause, and fixing the rate-of-fire defect. For the Colorpicker task, the steps were: successful fault reproduction, written root cause, correct test form, identifying a buggy input, and verifying the test.

We measured the time on task as the duration from the start of the initial reproduction attempt until the task was completed or until the participant ran out of time. We recorded the count and duration of all reproduction activities and whether the activity was mediated by Timelapse (automatic reproduction) or not (manual reproduction). Reproduction times only included time in which participants' attention was engaged on reproduction, which we determined by observing changes in window focus, mouse positioning, and interface modality.

Results

Below, we summarize our findings of how Timelapse affects developers' reproduction behavior (RQ1) and how this interacts with debugging expertise (RQ2).

Timelapse did not reduce time spent reproducing behaviors. There was no significant difference in the percentage of time spent reproducing behaviors across conditions and tasks. Though Timelapse makes reproduction of behaviors simpler, it does not follow that this fact will reduce overall time spent on reproduction. We observed the opposite: because reproduction with Timelapse was so easy, participants seemed more willing to reproduce behaviors repeatedly. A possible confound is that behaviors in our tasks were fairly easy to reproduce, so Timelapse only made reproduction less tedious, not less challenging. We had hoped to test whether Timelapse is more useful for fixing more challenging bugs, but were forced to reduce task difficulty so that we could retain a

within-subjects study design while minimizing participants' time commitment.

8–25% of time was spent reproducing behavior. Even when provided detailed and correct reproduction steps, developers in both conditions spent up to 25% (and typically 10–15%) of their time reproducing behaviors. Participants in all tasks and conditions reproduced behavior many times (median of 22 instances) over small periods. This suggests that developers frequently digress from investigative activities to reproduce program behavior. These measures are unlikely to be ecologically valid because most participants did not complete all tasks, and time spent on reproduction activities outside of the scope of our study tasks (i.e., during bug reporting, triage, and testing) is not included.

Expert developers incorporated replay capabilities. High-performing participants—those who successfully completed the most task steps—seemed to better integrate Timelapse's capabilities into their debugging workflows. Corroborating the results of previous studies [26, 22], we observed that successful developers debugged methodically, formed and tested debugging hypotheses using a bisection strategy, and revised their assessment of the root cause as their understanding of the defect grew. They quickly learned how to use Timelapse to facilitate these activities. They used Timelapse to accelerate familiar tasks, rather than redesigning their workflow around record/replay capabilities. In the Colorpicker task, participant 11 used Timelapse to compare program state before and after each call in the `mousemove` event handler, and then used Timelapse to move back and forth in time when bisecting the specific calls that caused the widget's RGB values to update incorrectly. Participants in the control condition appeared to achieve the same strategy more slowly by interleaving changes to breakpoints and manual reproduction.

Timelapse distracted less-skilled developers. Those who only achieved partial or limited success had trouble integrating Timelapse into their workflow. We partially attribute this to differences in participants' prior debugging experiences and strategies. The less successful participants used ad-hoc, opportunistic debugging strategies; overlooked important source code or runtime state; and were led astray by unverified assumptions. Consequently, even when these developers used Timelapse, they did not use it to a productive end.

Summary In our study, developers used Timelapse to automatically reproduce program behavior during debugging tasks, but this capability alone did not significantly affect task times, task success, or time spent reproducing behaviors. For developers who employed systematic debugging strategies, Timelapse was useful for quickly reproducing behaviors and navigating to important program states. Timelapse distracted developers who used ad-hoc or opportunistic strategies, or who were unfamiliar with standard debugging tools. Timelapse was used to accelerate the reproduction steps of existing strategies, but did not seem to affect strategy selection during our short study. As with any new tool, it appears that some degree of training and experience is necessary to fully exploit the tool's benefits. In our small study, the availability of Timelapse had no statistically significant effects on participants' speed or success in completing tasks. Figure 8 shows task success and task time per task and condition. In future work, we plan to study how long-term use of Timelapse during daily development affects debugging strategies. We also plan to investigate how recordings can improve bug reporting practices and communication between bug reporters and bug fixers.

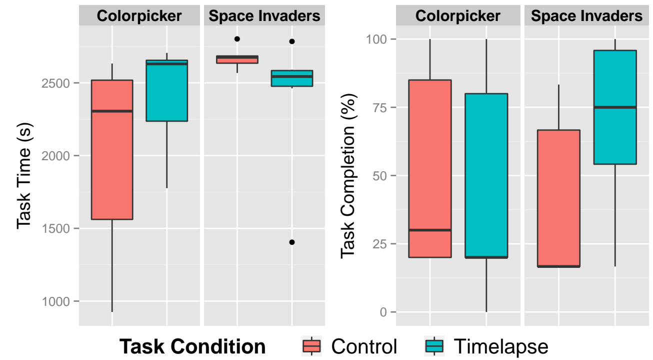

Figure 8. A summary of task time and success per condition and task. Box plots show outliers (points), median (thick bar), and 1st and 3rd quartiles (bottom and top of box, respectively). There was no statistically significant difference in performance between participants who used standard tools and those who had access to Timelapse.

RELATED WORK

Visualizing and Exploring Recordings and Traces

In contrast to the dearth of interactive record/replay tools, there have been many tools [12, 24] to visualize, navigate, and explore execution traces[8] generated by logging instrumentation. This is because execution traces are several orders of magnitude larger and contain very low-level details: the only way to understand them is to use elaborate search, analysis, and visualization tools. While Timelapse visualizes an execution as the temporal ordering of its inputs, trace-based debugging tools [12, 32] infer and display higher-level structural or causal relationships observed during execution. Timelapse's affordances primarily support navigation through the recording with respect to program inputs, while trace-based tools focus directly on aids to program comprehension (such as supporting causal inference or answering questions about what happened [12]).

Unfortunately, the practicality of many trace-based debugging tools is limited by the performance of modern hardware and the size of generated execution traces. Modern profilers, logging, and lightweight instrumentation systems [2] (and visualization tools [32] built on top of them) have acceptable performance because they capture infrequent high-level data or perform sampling. In contrast, heavyweight fine-grained execution trace collection introduces up to an order of magnitude slowdown [12, 20]. Generated traces and their indices [23, 24] are very large and often limited by the size of main memory.

Supporting Behavior Reproduction and Dissemination

To the best of our knowledge, deterministic record/replay systems that support dissemination of behaviors have only been

[8] *Execution traces* consist of intermediate program states logged over time, while Dolos's recordings consist only of the program's inputs.

widely deployed as part of video game engines [6]. Recordings of gameplay are artifacts shared between users for entertainment and education. These recordings are also a critical tool for debugging video game engines and their network protocols [30]. In the wider software development community, bug reporting systems [8] and practices [36] emphasize the sharing of evidence such as program output (e.g., screenshots, stack traces, logs, memory dumps) and program input (e.g, test cases, configurations, and files). Developers investigate bug reports with user-written reproduction steps.

While we have focused on the utility of record/replay systems for debugging, such systems are also useful for creating and evaluating software. Prior work has used record/replay of real captured data to provide a consistent, interactive means for prototyping sensor processing [3, 19] and computer vision [10] algorithms. More generally, macro-replay systems for reproducing user [31] and network [29] input are used for prototyping and testing web applications and other user interfaces. Dolos recordings contain a superset of these inputs; it is possible to synthesize a macro (i.e, automated test case) for use with other tools. The JSBench tool [25] uses this strategy to synthesize standalone web benchmarks. Derived inputs may improve the results of state-exploration tools such as Crawljax [17] by providing real, captured input traces.

CONCLUSION AND FUTURE WORK

Together, Timelapse and Dolos constitute the first toolchain designed for interactively capturing and replaying web application behaviors during debugging. Timelapse focuses on browsing, visualizing, and navigating program states to support behavior reproduction during debugging tasks. Our user study confirmed that behavior reproduction was a significant activity in realistic debugging tasks, and Timelapse assisted some developers in locating and automatically reproducing behaviors of interest. The Dolos infrastructure uses a novel adaptation of instruction-counting record/replay techniques to reproduce web application behaviors. Our prototype demonstrates that deterministic record/replay can be implemented within browsers in an additive way—without impacting performance or determinism, impeding tool use, or requiring configuration—and is a platform for new debugging aids.

Prior work assumes that executions are in short supply during debugging, and that developers know a priori what sorts of analysis and data they want before reproducing behavior. In future work, we want to disrupt this status quo. On-demand replay (in the foreground, background, or offline) could change the feasibility of useful program understanding tools [12] or dynamic analyses [4] that, heretofore, have been considered too expensive for practical (always-on) use. Using the Dolos infrastructure, we intend to transform prior works in dynamic analysis and trace visualization into on-demand, interactive tools that a developer can quickly employ when necessary. We believe that when combined with on-demand replay, post mortem trace visualization and program understanding tools will become *in vivo* tools for understanding program behavior at runtime.

ACKNOWLEDGEMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-0952733 and CCF-1153625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. Andrica, S., and Candea, G. WaRR: A tool for high-fidelity web application record and replay. In *DSN* (2011).
2. Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *USENIX ATC* (2004).
3. Cardenas, T., Bastea-Forte, M., Ricciardi, A., Hartmann, B., and Klemmer, S. R. Testing physical computing prototypes through time-shifted and simulated input traces. In *UIST* (2008).
4. Chow, J., Garfinkel, T., and Chen, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC* (2008).
5. Cornelis, F., Georges, A., Christiaens, M., Ronsse, M., Ghesquiere, T., and Bosschere, K. D. A taxonomy of execution replay systems. In *SSGRR* (2003).
6. Dickinson, P. Instant replay: Building a game engine with reproducible behavior. In *Gamasutra* (July 2001).
7. Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 211–224.
8. Glerum, K., Kinshuman, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., and Hunt, G. Debugging in the (very) large: Ten years of implementation and practice. In *SOSP* (2009).
9. Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).
10. Kato, J., McDirmid, S., and Cao, X. DejaVu: Integrated support for developing interactive camera-based programs. In *UIST* (2012).
11. King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC* (2005).
12. Ko, A. J., and Myers, B. A. Extracting and answering why and why not questions about Java program output. *ACM Trans. Softw. Eng. Methodol. 20*, 2 (Sept. 2010), 4:1–4:36.
13. Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng. 32*, 12 (Dec. 2006), 971–987.
14. Kuhn, A., and Greevy, O. Exploiting the analogy between traces and signal processing. In *ICSM* (2006).

15. Leshed, G., Haber, E. M., Matthews, T., and Lau, T. CoScripter: Automating and sharing how-to knowledge in the enterprise. In *CHI* (2008).
16. Mesbah, A., and van Deursen, A. Invariant-based automatic testing of AJAX user interfaces. In *ICSE* (2009).
17. Mesbah, A., van Deursen, A., and Lenselink, S. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB) 6*, 1 (2012), 3:1–3:30.
18. Mickens, J., Elson, J., and Howell, J. Mugshot: deterministic capture and replay for JavaScript applications. In *NSDI* (2010).
19. Newman, M. W., Ackerman, M. S., Kim, J., Prakash, A., Hong, Z., Mandel, J., and Dong, T. Bringing the field into the lab: Supporting capturing and RePlay of contextual data for design. In *UIST* (2010).
20. O'Callahan, R. Efficient collection and storage of indexed program traces, 2006. `http://www.ocallahan.org/Amber.pdf`.
21. Oney, S., and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *VL/HCC* (2009).
22. Parnin, C., and Orso, A. Are automated debugging techniques actually helping programmers? In *ISSTA* (2011).
23. Pothier, G., and Tanter, E. Summarized trace indexing and querying for scalable back-in-time debugging. In *ECOOP* (2011).
24. Pothier, G., Tanter, E., and Piquer, J. Scalable omniscient debugging. In *OOPSLA* (2007).
25. Richards, G., Gal, A., Eich, B., and Vitek, J. Automated construction of JavaScript benchmarks. In *OOPSLA* (2011).
26. Robillard, M. P., Coelho, W., and Murphy, G. C. How effective developers investigate source code: an exploratory study. *IEEE Trans. Softw. Eng. 30*, 12 (Dec. 2004), 889–903.
27. Saito, Y. Jockey: a user-space library for record-replay debugging. In *AADEBUG* (2005).
28. Sillito, J., Murphy, G. C., and Volder, K. D. Asking and answering questions during a programming change task. *IEEE Trans. Soft. Eng. 34* (2008), 434–451.
29. Telerik. Fiddler web debugging proxy, 2013. `http://www.fiddler2.com/fiddler2/`.
30. Terrano, M., and Bettner, P. 1500 Archers on a 28.8: Network programming in Age of Empires and beyond. In *Gamasutra* (March 2001).
31. The Selenium Project. Selenium WebDriver documentation, 2012. `http://seleniumhq.org/docs/03_webdriver.html`.
32. Trümper, J., Bohnet, J., and Döllner, J. Understanding complex multithreaded software systems by using trace visualization. In *SOFTVIS* (2010).
33. VMWare, Inc. Replay debugging on Linux, October 2009. `http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf`.
34. Yeh, T., Chang, T.-H., and Miller, R. C. Sikuli: Using GUI screenshots for search and automation. In *UIST* (2009).
35. Zaidman, A., Matthijssen, N., Storey, M.-A., and van Deursen, A. Connecting client and server-side execution traces. *Empirical Software Engineering (EMSE) 18*, 2 (2013), 181–218.
36. Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schrter, A., and Weiss, C. What makes a good bug report? *IEEE Trans. Soft. Eng. 36*, 5 (September 2010), 618–643.