On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications

Mouna Hammoudi[1], Brian Burg[2], Gigon Bae[1], Gregg Rothermel[1]
[1]University of Nebraska - Lincoln, USA *{mouna,gbae,grother}@cse.unl.edu*
[2]University of Washington, USA *burg@cs.washington.edu*

ABSTRACT

Recording the sequence of events that lead to a failure of a web application can be an effective aid for debugging. Nevertheless, a recording of an event sequence may include many events that are not related to a failure, and this may render debugging more difficult. To address this problem, we have adapted Delta Debugging to function on recordings of web applications, in a manner that lets it identify and discard portions of those recordings that do not influence the occurrence of a failure, We present the results of three empirical studies that show that (1) recording reduction can achieve significant reductions in recording size and replay time on actual web applications obtained from developer forums, (2) reduced recordings do in fact help programmers locate faults significantly more efficiently as, and no less effectively than non-reduced recordings, and (3) recording reduction produces even greater reductions on larger, more complex applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Reliability, Experimentation

Keywords

Delta Debugging, Web Applications, Record/Replay Techniques, Recording Reduction

1. INTRODUCTION

Users of web applications often experience intermittent, annoying failures that reduce their productivity. When such a failure is encountered, a user may be prompted to send feedback to the application's developers. While better than nothing, user-provided bug reports frequently omit important details such as clear reproduction steps and expected behavior [55]. Many crash reporting tools supplement user reports with automatically gathered post-failure data (e.g., crash stacks). This is not without its costs: developers can easily be overwhelmed by the large amount of collected data [14]. Even with detailed post-failure reports, it can be difficult to locate a fault because these reports often lack the specific inputs and execution conditions underlying the failure.

An alternative approach for reporting a failure is to automatically capture the inputs and events that led to it. Macro-replay tools such as Sikuli [51], CoScripter [28], and Selenium [47] do this, reproducing user inputs for automated testing or sharing automated workflows. Researchers have also been investigating record/replay approaches for web applications and Javascript programs [2,32,48]. There has also been research on deterministic record/replay techniques [7,12], that capture entire executions while additionally controlling sources of nondeterminism (Section 6 provides details).

A record/replay infrastructure for web applications captures user inputs and events (mouse, keyboard, navigation commands, etc.) before they are hit-tested or parsed by the browser engine. During playback, inputs and events are re-delivered to the browser engine. A challenge faced by record/replay approaches is that the recordings they capture may be lengthy and thus, difficult to utilize. Even if an execution is exactly recorded, the number of events and program states can render opportunistic debugging strategies ineffective [7]. Captured inputs and events that are not necessary to reproduce a failure are distracting, yet no prior work has attempted to reduce the size of recordings of web applications.

In this work we adapt the well-known Delta Debugging algorithm [11, 53] to function on recordings of web applications, in a manner that lets it identify and discard portions of those recordings that do not influence the occurrence of a failure. When a user encounters a failure in a web application, he or she can record an extended interaction with the application and send it to the application's developers. The user's recording can then be reviewed and reduced, and utilized in bug triage, fault localization, and regression testing. Alternatively, developers can capture recordings of web applications found to contain faults in-house, and reduce these recordings in order to find and correct the faults.

There are several questions that one might wish to answer about the effectiveness and the efficiency of recording reduction via Delta Debugging in the domain of web applications, but three questions that we believe are particularly important are (1) whether the approach sufficiently reduces recordings, (2) whether reduced recordings actually facilitate debugging, and (3) whether the approach scales to large and complex web applications. To address these questions, we conducted three empirical studies.

In our first study, we sought to determine whether recording reduction via Delta Debugging can in fact produce potentially effective reductions in recordings when applied to a relatively large number of actual web applications containing actual failures re-

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786846

ported by their programmers. The study assessed the degree of reduction that could be achieved by applying recording reduction to 30 faulty web applications obtained from developer forums. The technique was able to reduce each recording to 50% or less of its original size, on average reducing recording sizes to 17.1% of original sizes, and reducing replay times to 34.6% of original times.

Years of work on fault localization techniques have resulted in numerous empirical studies showing that these techniques can reduce the numbers of statements that could be seen as related to faults. Virtually all papers on this topic have simply assumed that this would help programmers locate faults, without directly studying whether it could in fact do so. Parnin and Orso [42] recently provided data suggesting the potential folly of that assumption. Over 15 years of work on Delta Debugging have resulted in empirical studies that show that it can reduce the sizes of input sets or programs, but here as well, we can find no published studies of whether this can help programmers, either. Thus, in our second study we asked twenty computer science students to detect failures, locate faults, and correct them given reduced and unreduced recordings made over web applications. The results of this study show that reduced recordings statistically significantly decreased the time required by developers to locate and correct faults, without adversely affecting their effectiveness in doing so.

Results of our first and second studies show that Delta Debugging can substantially reduce recordings and facilitate the debugging process, but were limited to extracts of code posted by developers online. Our third study addresses scalability issues, attempting to ascertain whether Delta Debugging can successfully reduce recordings of larger, more complex web applications. Thus, in our third study we applied our recording reduction technique to several larger, more complex web applications. The technique was able to reduce each recording to 28% or less of its original size, on average reducing recording sizes to 14.0% of original sizes, and reducing replay times to 22.2% of original times.

2. DELTA DEBUGGING FOR WEB APPS

In this section, we describe our adaptation of Delta Debugging for reducing recordings of web application failures. We then discuss some simplifying assumptions made to facilitate the work.

2.1 Algorithm and Implementation

Selenium and other replay tools enable a user to capture a recording R of events that lead to the occurrence of a specific failure f. Our goal is to reduce recording R to a new recording R' by discarding events[1] that are not required for failure f to occur. Using R' to reproduce f, a developer should be able to consider fewer events and less code when attempting to locate the fault responsible for f.

Our recording reduction approach adapts the Delta Debugging algorithm presented in Reference [53] to operate on recordings of web applications. At a high level, the algorithm operates by repeatedly selecting subsets of the events in a recording, and replaying these on the failing application to determine whether these subsets can, by themselves, reveal the failure. If so, then the algorithm continues to search these subsets for smaller failure-revealing subsets, until no such subsets can be found.

More precisely, the Delta Debugging algorithm attempts to partition a given recording R' into a number of subsets and test each subset as well as its complement subset until each subset contains only one change. If the execution of a subset raises failure f, the algorithm treats the failing subset of R' as a new recording and repeats the same procedure until there is no smaller subset causing failure f. Numbers of subsets are determined by a variable n that stands for "granularity"; n is initialized to two at the first iteration but subsequently can change (1) to two if a subset induces failure f ("reduce to subset"), (2) to $max(n-1,2)$ if a subset's complement induces failure f ("reduce to complement"), or (3) to $min(|R'|, 2n)$ if $n < |R'|$ ("increase granularity"). Otherwise, the reduction procedure terminates and returns R', a *1-minimal test case* [53] from which no one element can be removed while preserving failure f.

A critical building block for any recording reduction algorithm is a *failure oracle*: the ability to tell whether the failure occurs during an execution. Our approach does not depend on any particular oracle; for the purposes of our studies we use manually inserted assertions that detect whether a failure occurs. Given a web application with a known failure, such assertions can be inserted by software engineers prior to proceeding with recording reduction.

Our implementation makes use of recordings generated by Selenium. We chose this platform for several reasons, including the facts that it is open-source, is widely used among web application testers with regard to web application testing, and there is a wide availability of IDEs and abundant web browser control APIs that make it easy to record event sequences as a test case and to execute them programmatically. Selenium-IDE, a Firefox extension, can record a user's actions (or events) in the Firefox browser and save them as a record (or a test case) in various formats (e.g., HTML, Java JUnit/TestNG, Ruby RSpec, and C# NUnit). Our prototype implementation is written in Java, and uses the Selenium WebDriver library[2] and Groovy script engine; this allows the tool to execute any portion of a Selenium script.

The input to our tool is a test script derived from the recording made with the Selenium IDE. The recording reflects the user's interactions with the interface. Each time the user interacts with the interface, a Selenese command is inserted into the recording; once the recording process is completed, the final set of Selenese commands forms a test script. Each Selenese command represents one user interaction with the interface and is denoted by the following tuple: <action, locator, value>. The action component of a Selenese command indicates the event that is performed on the user interface (e.g., click, select) during the recording process. The locator component specifies the user interface element (i.e., input field) that the user is interacting with during the recording process. The value component refers to any input entered by the user within the locator previously specified, while recording user interactions. The first line of every Selenium test script consists of the `open` Selenese command, which specifies the URL of the web page under test. The action of the `open` command is "open" and its locator is that URL. We inserted assertions within our test scripts via the Selenium IDE to signal the occurrence of a failure during the replay process. Once the recording process ends, the test script is exported via the Selenium IDE as a "JUnit4 WebDriver" script. Our recording reduction tool is able to replay the user interactions initially recorded based on the sequence of Selenese commands contained within this test script.

Our approach depends on the ability to replay subsections of a failure recording. In general, Delta Debugging can generate reduced recordings that are infeasible, in that they contain events that cannot be executed. Selenium can start playback from any position in a recording, but if the recording ends up being infeasible, our im-

[1]We use the term event to define the various elements that appear in Selenium traces, such as keyboard entries inputs, and mouse movements; in general, however our approach is applicable to most varieties of event- and input-based replay systems.

[2]http://docs.seleniumhq.org/download/

plementation of the playback follows the approach used by Delta Debugging, and outputs "UNRESOLVED" as the test result of the subset. Otherwise, each test of a subset outputs "PASS" or "FAIL", according to the existence of failure f. Future work could analyze dependencies among events and filter out some non-feasible sequences before executing them, but whether this would reduce the cost of the approach would need to be studied.

Finally, because recordings of web applications may include certain steps that *must* be present in order to replay any portion of them (e.g., the input of a username and password), our implementation also allows users to flag certain events in a Selenium trace as *prefix events*; all such events are ignored during recording reduction, ensuring that they remain in any reduced recording.

Given that data sources could change between a recording session and a replay session, it is possible that some of our test cases could be rendered obsolete. To address this issue, our implementation accepts a URL whose HTTP request is established before executing the *prefix events*. This feature can be used to invoke other initialization tasks that need to be performed prior to running any subset of events during Delta Debugging. For example, in Study 3, where our object programs utilize databases, we use this feature to allow the web application server to rollback to its initial database state and user session state prior to running any event sequence; otherwise, replay results could be adversely affected.

To construct our implementation of Delta Debugging we utilized code made available by Zeller,[3] that is based on the algorithm published in [53].[4] Since the algorithm does not guarantee that each subset is tested only once, it may test the same subset multiple times. We use a cache to store test outcomes to improve efficiency.

2.2 Simplifying Assumptions and Limitations

To function correctly, our algorithm depends on the ability of Selenium to deterministically reproduce a web application's behavior. Javascript itself is a single-threaded language [30, 46], but Javascript programs can indeed face sources of non-determinism. While Selenium can often control for such non-determinism, particularly when the application being debugged is under the control of the programmer doing the debugging, this may not always be the case. In such cases, we would need to resort instead to the use of deterministic record/replay techniques [12], which, as noted in Section 1, include approaches that capture entire executions by additionally controlling sources of nondeterminism. One such approach – also noted in Section 1 and described in greater detail in Section 6, is that of Burg et al. [7]. As infrastructures such as these mature, they should allow applications of reduction algorithms such as ours in scenarios where tools like Selenium do not suffice.

The Delta Debugging algorithm that we utilize relies on trial and error rather than precise tracking of runtime data dependencies to decide which inputs to discard. Data dependencies between tasks are treated as black boxes; with better data dependency tracking, more inputs could be safely discarded. Tools such as the WhyLine for Java [21] have had great success in leveraging dynamic slicing algorithms to identify a minimal causal chain of events leading up to a specific output or program state. Still, as our subsequent studies show, the algorithm we have presented can be quite effective at reducing recording sizes.

Failure oracles are also critical to the efficacy of our approach. For example, a naive "diff" oracle might compare the outputs of R and R' and detect that their outputs differ. However, the difference in outputs may have nothing to do with the failure's manifestation. As noted earlier, in this work we depend on engineers to insert assertions that can indicate whether a particular, known failure recurs. However, given the varied modalities of web application failures, we expect many types of oracles to be applicable in this context. Prior work has focused on text- and JavaScript-centric oracles [36], but interactivity oracles [22] and visual oracles [51] may also be useful. In particular, oracles that can identify transition instants—the statement or event where a failure manifests—are helpful for isolating failures to specific intervals of a recording. Transition-revealing oracles depend on the capabilities of the underlying replay tool or execution environment. Arya et al. [5] discuss practical issues in detecting such transitions, such as non-monotonicity of transition instants and integration with breakpoint debuggers.

Finally, Delta Debugging has certain inherent limitations that our implementation of it shares. When a program contains multiple faults, it may be difficult to correctly assess whether a particular failure is being reproduced by a subset of a recording, as opposed to a failure prompted by some other cause. Somewhat similarly, an attempt to assess whether a failure reoccurs given a subset of a recording may suffer from what we might call "coincidental incorrectness": a case in which an oracle incorrectly deduces that a given output is incorrect and does represent the recurrence of a failure. In our empirical studies, we were able to determine that the second limitation did not surface in the cases that we considered, but in the general case they remain possible.

3. STUDY 1: FEASIBILITY

Our first study examines the following research question:[5]

RQ1: To what extent can Delta Debugging reduce recordings of web applications?

3.1 Objects of Analysis

We chose Stack Overflow as a source for objects of analysis in this study. In part this choice was made due to the popularity of this forum among web application developers. As of March 2015, Stack Overflow contained 388,394 questions related to HTML, 797,254 questions related to JavaScript and 286,167 questions related to CSS. We randomly selected 30 objects of analysis from those that developers had posted within Stack Overflow, restricting our selection to cases in which the developers had made both code, and replicable descriptions of failures in that code, available. Table 1 provides basic data on the objects of analysis, along with their functionality, size and type of faults contained, and citations that indicate where they can be found.

In this study, our objects of study consist of client side code embedding faults at the level of HTML, JavaScript and CSS. One might argue that these objects of study are relatively short code extracts that do not reflect the complexity of typical web applications. Nevertheless, these code extracts were initially posted by developers on Stack Overflow, which indicates that their developers indeed had difficulties locating and correcting the faults by themselves. Further, our objects of study had generated an average of five responses by Stack Overflow administrators, which also attests to the non-trivial nature of the faults under consideration. In addition, the objects of study were all developed by different programmers, and were diverse in the types of functionality they provide.

[3]http://www.whyprogramsfail.com/resources.php

[4]Zeller's implementation omits one efficiency-improving step ("reduce to subset") of the algorithm published in Reference [53], that does not impact its effectiveness but increases its efficiency, and we omit this step as well.

[5]Experiment data (including Selenium recordings, transformed test scripts and delta-debugging logs) for Studies 1 and 3 is publicly available at https://sites.google.com/site/fsewebdd.

Table 1: Objects of Analysis

#	Functionality	LOC	Fault Type
1	Host travellers [57]	136	Invalid input
2	Applicant survey [68]	144	Checking more checkboxes than allowed
3	Apply for a passport [79]	78	Missing textbox
4	Create an account [82]	86	Inappropriate dialog box output when the user clicks "submit"
5	Calculate age in days/hours/min [83]	114	Invalid formula for calculation
6	Create an account [84]	115	Inappropriate error message
7	Play with numbers [85]	64	Invalid input
8	Shopping cart [86]	30	Cannot add to items in cart
9	Game: Reorder letters in word [87]	68	Faulty button click
10	Pool tournament [58]	36	Unable to delete player from team
11	Filling complaint form [59]	49	Unresponsive to button click
12	Submit a form to win a prize [60]	50	Header not scrollable
13	Schedule an appointment online [61]	134	Weekends not disabled
14	Prepare for a concert [62]	56	Unresponsive page
15	Book a trip [63]	105	Incorrect values listed in dropdown menu
16	Gather ingredients for a cooking workshop [64]	42	Incorrect behavior
17	Best used cars in town [65]	38	Invalid Input
18	Course listings [66]	150	Invalid form layout
19	Borrow a book from a library's website [67]	115	Invalid input
20	Airport pickup information [69]	44	Allowed form submission even if the user enters a phone number in the email text field
21	Rent a car [70]	125	Unresponsive DatePicker
22	Veterinary form [71]	130	Inverted disable/enable input fields
23	Dentist form [72]	86	Invalid email address
24	Waiting list for classes [73]	92	Invalid input
25	Playing with numbers [74]	118	Wrong value output due to incorrect calculation
26	Phone company customer survey [75]	117	Invalid form layout
27	Patient form [76]	93	Form submitted with empty fields
28	Insurance account creation [77]	93	Two fields left blank
29	Financial aid form [78]	99	Inappropriate dialog box
30	Online shopping [80]	99	Invalid amount input

3.2 Variables

Our independent variable consists of the type of recording utilized (reduced or unreduced).

As dependent variables, we measured the sizes of full and reduced recordings in terms of the numbers of events, and the time required to replay the full and reduced recordings.

3.3 Study Setup and Design

To obtain unreduced recordings for each web application, we found a sequence of events which, when applied to that web application, reproduced the failure that had been reported for it. Usually, such a sequence had been reported on Stack Overflow as part of the failure report. For example, one person reported: "The following code is supposed to display a confirmation box. If the user clicks the ok button, the current time is supposed to be displayed. But for some reason when I test the button by clicking on it nothing happens. I need a fresh pair of eyes to tell me what I'm missing as to why this code isn't working." We then applied that sequence of events with Selenium operating to capture the unreduced recording. Next, we inserted assertions into the applications (as described in Section 2) to allow the failures to be observed, and we applied our Delta Debugging implementation to the unreduced recordings. To obtain recording replay time we reran Selenium on both the original and reduced recordings. We executed this experiment, as well as those reported for Study 3, on an OS X 10.10 MacBook Pro with 2.6 GHz Intel Core i5 and 16GB of RAM memory.

3.4 Threats to Validity

The primary threat to external validity for this study involves our reliance on relatively small client side web pages rather than more complex web applications. Through Stack Overflow we had access to open source code and failure reports only for web pages extracted from web applications. Thus, our study does not take into consideration all of the complexities of modern web applications (a limitation that our third study seeks to address). However, all of the pages that we consider are derived from real applications, with actual reported faults, albeit, with only a single fault per application. A threat to internal validity involves the possibility of faults existing in our Delta Debugging implementation, but this is mitigated to some extent by the fact that we confirmed that both our unreduced and reduced recordings did in fact reveal the same failures.

3.5 Results and Analysis

Table 2 addresses RQ1. The table quantifies the differences in size and replay time observed before and after reduction. As the data shows, each recording was reduced to 50% or less of its original size. The average size of full recordings was 13.1 events while the average size of reduced recordings was 1.9 events (i.e., on average recording sizes were reduced to 17.1% of the sizes of unreduced recordings). The average replay time for full recordings was 6.1 seconds, while the average replay time for reduced recordings was 1.8 seconds (i.e., on average replay times were reduced to 34.6% of the replay times of unreduced recordings).

To determine whether Delta Debugging produced statistically significant reductions in recording size and replay time, we conducted two-tailed Mann-Whitney U-tests [24] at significance level $\alpha = 0.05$. Our null hypotheses were "Recording reduction does not reduce recording size" and "Recording reduction does not reduce replay time". The tests indicated that the reductions in size and replay time were both statistically significant, with p-values 0.000002 and 0.000002, respectively.

3.6 Discussion

The data obtained in our study demonstrates the potential effectiveness of our Delta Debugging algorithm applied to web applications – at least, of the class considered in our objects of study. The approach was able to substantially (and statistically significantly) reduce recording size and replay time. Potentially, such reduced recordings can help programmers debug web applications more cost-effectively.

4. STUDY 2

Our second study considers the following research question:

RQ2: To what extent does a reduced recording assist programmers in the debugging process given a faulty web application?

4.1 Participants

We recruited 20 participants from the population of graduate and undergraduate students enrolled in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln, by sending an email request to all such students. From those who responded, we excluded persons who lacked experience with HTML, JavaScript and CSS, and then we selected the first 20 respondents

Table 2: Recording Size, Replay Time Before/After Reduction

#	Size (events)			Replay Time (seconds)		
	Full Recording	Reduced Recording	Pct.	Full Recording	Reduced Recording	Pct.
1	13	2	15.4	5.2	2.2	42.3
2	9	2	22.2	3.5	1.4	40.0
3	10	1	10.0	4.9	1.4	28.6
4	18	2	11.1	6.4	1.6	25.0
5	9	3	33.3	7.6	5.9	77.6
6	50	1	2.0	19.1	1.1	5.8
7	11	2	18.2	4.0	1.7	42.5
8	14	1	7.1	12.0	1.7	14.2
9	14	1	7.1	4.9	1.3	26.5
10	8	1	12.5	2.7	1.2	44.4
11	13	2	15.4	7.0	1.8	25.7
12	11	1	9.1	4.3	1.2	27.9
13	30	6	20.0	12.9	6.7	51.9
14	8	2	25.0	3.7	1.6	43.2
15	17	2	11.8	6.9	2.0	29.0
16	30	1	3.3	22.2	1.6	7.2
17	4	2	50.0	2.1	1.5	71.4
18	10	1	10.0	4.6	1.1	23.9
19	14	2	14.3	5.9	1.4	23.7
20	6	2	33.3	3.3	1.8	54.5
21	11	3	27.3	4.2	1.5	35.7
22	13	2	15.4	4.9	1.4	28.6
23	7	1	14.3	3.2	1.3	40.6
24	9	1	11.1	3.7	1.1	29.7
25	8	2	25.0	4.2	1.5	35.7
26	11	1	9.1	4.1	1.1	26.8
27	9	2	22.2	3.7	1.2	32.4
28	10	2	20.0	5.0	1.9	38.0
29	6	1	16.7	3.6	1.2	33.3
30	10	2	20.0	4.0	1.3	32.5

from those who remained eligible. Participants were each offered \$20 in compensation for their time.

We used a survey to obtain demographic information about participants. Thirteen participants were graduate students and seven were undergraduates; all were male. On a 5-point Likert scale, six participants rated their experience with web programming as "average", seven as "good", six as "very good" and one as "excellent". Participants had a mean of 3.05 years and standard deviation of 2.4 years experience programming web applications using HTML, JavaScript and CSS. 11 participants had learned web programming by themselves, four through courses and five through internships. Seven participants were familiar with record and replay infrastructures, but only four had used Selenium. The mean number of web applications participants had developed was 5.8 with a standard deviation of 1.7. 12 participants rated their debugging skills as "good", six as "very good" and two as "excellent".

4.2 Variables

Our independent variable consists of the type of recording utilized (reduced or unreduced).

We considered two dependent variables. Our first dependent variable relates to efficiency, and measures the wall clock time required by each participant for a given faulty web application to (1) detect the failure, (2) locate the fault responsible for the failure, and (3) correct the fault. Note that in some cases, participants were unable to locate and/or correct faults, and in those cases, times required to achieve these goals are not considered. The second variable relates to effectiveness, and measures whether or not each participant succeeded, for a given faulty web application, in doing each of these three things.

4.3 Study Setup and Design

We selected four web applications. These included a tic tac toe game application of 707 LOC in which the fault involved omitted code [81], and the applications numbered 4, 20 and 21 in Table 1. Here, we refer to these as Applications A, B, C, and D, respectively. We chose these applications because they were of varying sizes ranging from relatively small (44 LOC) to much larger (707 LOC), and each involved different types of failures. We used unreduced Selenium recordings, along with reduced recordings derived by the Delta Debugging algorithm used in Study 1, with assertions removed.

We conducted the study in the Usability lab at the University of Nebraska-Lincoln. We organized twenty individual sessions of 90 minutes in order to observe each participant individually.

We began each session with a tutorial about Selenium. Next, we gave the participant a faulty web application with a statement of how it was expected to work, and a recording of the application's behavior in a case where it fails. We used this to illustrate the process of (1) replaying the recording, (2) identifying and recording information on the failure, (3) attempting to locate the underlying fault in the application, and (4) attempting to correct that fault. During this process we worked with the participant and answered questions, but we did not inform them about the purpose of the study or make any statements related to recording types or lengths.

When each participant was confident enough to proceed, we gave them the four faulty web applications, two with reduced and two with unreduced recordings. We counterbalanced the assignment of web applications across all 20 participants such that all possible combinations of web applications and recordings were utilized and distributed as evenly as possible. We also varied the orders in which participant considered web applications to lessen the impact of learning effects. With 20 participants each considering two unreduced and two reduced recordings, we were able to obtain 10 sets of results relative to each of the two reduced and unreduced recordings of the four web applications.

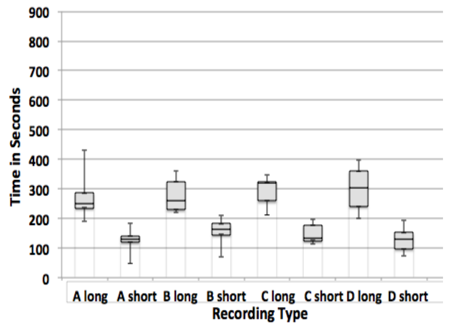
We asked each participant to run the recording for each web application, attempt to detect the failure within that application, and if successful at that, to attempt to locate the fault responsible for it and correct it. During this portion of the study we captured screen recordings using the usability testing software "Morae"; this enabled us to later revisit the debugging process conducted by each participant. We did not utilize a think-aloud methodology, because we wished to measure the time taken to perform tasks, and a think-aloud methodology would render these measurements problematic.

We limited the time allowed for each of the four applications to 15 minutes each, to keep the overall task manageable and limit possible fatigue effects. As the participants proceeded, we recorded the times at which they found failures, noted faults, and concluded that faults had been corrected. These times provide our data on efficiency. We also asked each participant to write a brief description of each failure located and each fault they were able to locate. Finally, we asked each participant to complete an exit survey to obtain further feedback and comments.

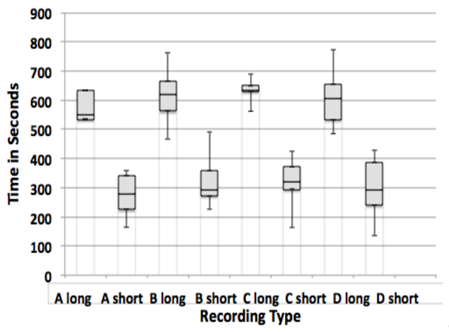
When all sessions were finished, we reviewed the participants' notes about failures and faults found, together with (when available) the version of the web application they pronounced correct, to determine whether they had correctly noted the failures and faults and applied an appropriate correction. The first author, together with a staff programmer not involved with this research, each performed this assessment individually, and met to compare notes and come to a consensus as to which results were indeed correct. This provided our data on effectiveness.

4.4 Threats to Validity

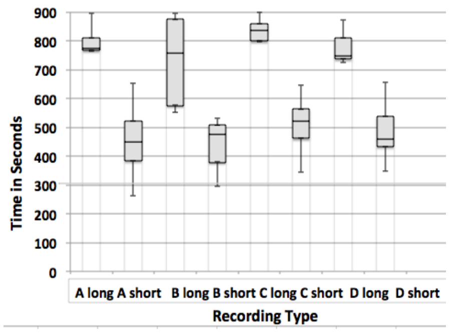
The primary threat to external validity for this study involves the objects of study utilized. We studied only four such objects,



(a) Failure Detection Time



(b) Fault Localization Time



(c) Fault Correction Time

Figure 1: Timing Results

all of which were client-side web pages. Still, all four objects of study are actual web applications containing actual faults which their original creators had difficulty correcting, so they do represent at least one sub-class of actual applications of interest. A second threat to external validity concerns our participants, all of whom were students, not practicing professional web application engineers. However, this group of participants does represent one actual, non-trivial class of persons who program web applications, and they are interesting in that respect. Threats to internal validity may involve learning effects; we guarded against these by counterbalancing and varying the orders of web applications and recordings. Threats to construct validity involve the use of a 15 minute time limit for debugging tasks; times longer than 15 minutes might have allowed participants to locate and correct additional faults. However, a limit needed to be chosen and as discussed in Section 4.6 it affected only five instances of fault localization (all on unreduced recordings) and 10 of fault correction (five on unreduced and five on reduced recordings). It seems likely that while longer limits might have affected differences in effectiveness, they would only have accentuated differences in efficiency.

4.5 Results

4.5.1 Efficiency

Figures 1.(a), 1.(b) and 1.(c) use boxplots to present the amount of time required by participants to detect failures, locate the underlying faults, and correct those faults for each web application, using both unreduced (".long") and reduced (".short") recordings. The horizontal axes list the web applications and recording types and the vertical axes indicate time required. The boxes thus indicate the distributions of times participants required to detect failures, locate faults, and correct faults, across the three graphs respectively. In the figures we use *cumulative* numbers to report the times required to locate a fault and correct it, because these provide a more appropriate understanding of the total times required to reach the various debugging task checkpoints. For example, for C.long, the mean time required to detect the failure was 296.4 seconds, the mean time to detect the failure and locate the fault that caused it was 626 seconds, and the mean time to detect the failure, locate the fault and correct it was 880 seconds. Also, as noted earlier, we report times only for cases in which participants were able to complete tasks. For example, the boxplot for D.long presents data for only the four participants who corrected the fault in that case.

Overall, using unreduced recordings, participants required longer times to detect failures, locate faults, and correct them. Considering failure detection, this is evident in the boxplots for all four applications. Across all four applications, unreduced recordings led to a mean failure detection time of 282.4 seconds with a standard deviation of 68.8. In contrast, reduced recordings led to a mean time of 142.9 seconds with a standard deviation of 31.2.

Table 3: Statistical Results: Confidence Intervals

Obj	Detect Failure	Locate Fault Cumul.	Fix Fault Cumul.	Locate Fault non-Cumul.	Fix Fault non-Cumul.
A	[244,315]	[596, 638]	[287, 289]	[297, 376]	[165, 285]
B	[71, 129]	[484, 626]	[695, 805]	[372, 520]	[132, 246]
C	[187, 254]	[542, 644]	[723, 828]	[384, 482]	[109 218]
D	[263, 342]	[167, 332]	[264, 465]	[271, 473]	[132, 203]

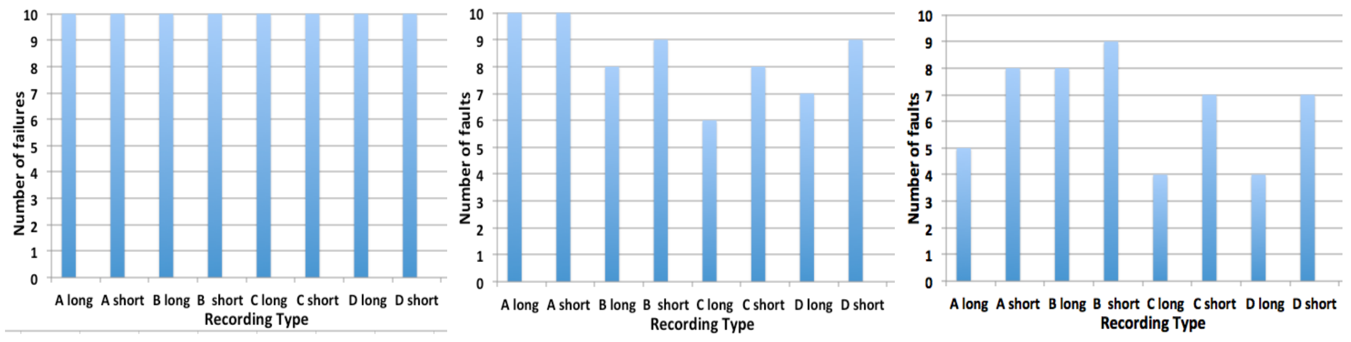
When the cost of fault localization is added to that of failure identification, using unreduced recordings, participants again required more time (in cases where they succeeded) than when using reduced recordings, on all four applications. Across all four applications, unreduced recordings led to a mean time of 603.2 seconds with a standard deviation of 84.5, whereas reduced recordings led to a mean time of 308.6 seconds with a standard deviation of 80.4.

When the cost of correcting faults is added on, unreduced recordings led to a mean time of 787.6 seconds with a standard deviation of 85.6. In contrast, reduced recordings led to a mean time of 471.4 seconds with a standard deviation of 102.1. Hence, the time required to repair faults was higher for unreduced recordings than reduced ones. However, the standard deviation was higher at the level of reduced recordings compared to unreduced ones: the amounts of time needed to repair faults were spread out over a larger range of values with reduced recordings than with unreduced ones.

To determine whether the differences in times were statistically significant, we conducted bootstrap tests [24] on the data obtained for each program's unreduced and reduced recordings. (We chose this test because it is applicable to non-normal data distributions and can be utilized on data sets of unequal sizes such as we have for most of our object web applications where fault localization and fault correction are concerned). Our null hypotheses were "Recording reduction does not increase programmer efficiency with respect to [failure detection | fault localization | fault correction] efficiency," and we tested these hypotheses for each of the four applications at a confidence level of 95%. In addition, for fault localization and fault correction, we examined this hypothesis on both cumulative times, and on the times required just for the specific tasks.

Table 3 presents the confidence intervals returned by the bootstrap tests. In all cases these confidence intervals do not cover 0 (and in all cases p-values were 0.04), allowing us to reject the null hypotheses in all cases.

Overall, reduced recordings statistically significantly increased programmers' efficiency in failure detection, fault localization and fault correction, considering time cumulatively or non-cumulatively.



(a) Failure Detection Correctness (b) Fault Localization Correctness (c) Fault Correction Correctness

Figure 2: Correctness Results

4.5.2 *Effectiveness*

Figures 2.(a), 2.(b) and 2.(c) present the numbers of failures detected, faults located and faults corrected by the participants, respectively, for each of the web applications, using both unreduced (".long") and reduced (".short") recordings. The horizontal axes list the web applications and recording types, and the vertical axes indicate numbers of faults/failures. The heights of bars indicate, for each application, the numbers of participants who detected failures, located faults, and repaired faults correctly, across the three graphs.

As Figure 2.(a) shows, all of the participants were able to detect each of the failures embedded within the applications they considered, regardless of recording size.

As Figure 2.(b) shows, all participants were able to locate the fault embedded within Application A. Applications B, C and D follow a different pattern: on these applications, participants were not always able to correctly locate the faults. However, they were correct more often when using the reduced recordings.

As Figure 2.(c) shows, where fault correction is concerned, differences in results across recording sizes were even more apparent. However, keep in mind that only participants who located the faults could correct them. On Application B, all participants who located the fault were able to correct it with both reduced and unreduced recordings, and on Application D, two participants who located the fault were not able to correct it with reduced or unreduced recordings. On the other hand, for A.long, five participants who located the fault were unable to correct it, while for A.short, only two who located the fault were unable to correct it, and for C.long, two participants who located the fault were unable to correct it, while for C.short, only one who located the fault could not correct it. This suggests that programmers' abilities to correct faults may be enhanced by reduced recordings.

Since our effectiveness data is categorical, and in two cases involves fewer than five data points, we used Fisher's exact test [24] to assess whether the differences in effectiveness, comparing unreduced and reduced recordings in the different tasks, were statistically significant. Our null hypotheses were "Recording reduction does not increase programmer effectiveness with respect to [failure detection | fault localization | fault correction] efficiency," and we tested these hypotheses for each of the four applications at a confidence level of 95%. In no case was statistical significance found. Thus, we cannot conclude that the numerical differences observed in effectiveness were necessarily caused by reductions in recording size. We *can* conclude, however, that programmers using short recordings were no less effective than those using long ones.

4.6 Discussion

We asked our participants to complete exit surveys to help us gain insights into their perceptions of the experience. 16 participants considered their background sufficient given the applications they used, while only four felt that their background was not sufficient. 11 students considered the overall level of difficulty of the web applications "average", five rated it "easy" and three considered it "difficult". Regarding the students' ratings of their overall experience locating faults and fixing bugs, 14 considered the task to be of "average" difficulty, three rated it as "easy" and three considered it "difficult". These numbers lead us to conclude that our participants and the web applications that we selected were, overall, appropriate for the study.

As previously noted, the task of correcting faults involved a higher standard deviation in efficiency results for reduced recordings than for unreduced ones. Figure 2.(c) reveals that participants tended to locate larger numbers of faults correctly when using reduced recordings. The data corresponds, however, only to correct repair efforts. Thus, the data represent larger numbers of values for reduced recordings than for unreduced recordings, increasing the chances for observing variance in that data.

We also noted previously that all the participants were able to locate the fault in Application A regardless of recording type (reduced or unreduced). However, participants were not all able to correctly locate the faults embedded in applications B, C and D. The fault embedded in Application A is related to a code omission that could easily be visible to the participants. However, the faults in Applications B, C and D were related to existing code that was incorrect in a manner that was not necessarily obvious. Therefore, locating faults in the latter applications was more challenging for participants.

Even though participants were all able to locate the fault in Application A, they were not all able to repair it, particularly when relying on Application A's unreduced recording. Due to the omission of code in A, the participants had to rewrite the omitted lines of code, which was a more error-prone operation than correcting lines of code that were already present but erroneous.

Participants required less time to locate faults when using reduced recordings, giving them more time to make progress towards comprehending and fixing the fault. Table 4 illustrates the number of participants who reached the 15 minute time limit allotted for the fault localization and fault correction stages of tasks. A comparison of Figure 2.(b) with Table 4 reveals that no participants ran out of time while attempting to locate faults given reduced recordings, though some were incorrect in their assessments of the faults. In contrast, several participants ran out of time when attempting to locate faults using unreduced recordings.

4.7 Analysis of Screen Recordings

The results of our second study show that Delta Debugging can substantially facilitate the debugging process by decreasing the time

Table 4: Current Activity as Task Time Limits Were Reached.

	A.L	A.S	B.L	B.S	C.L	C.S	D.L	D.S
Fault Localization	0	0	1	0	2	0	2	0
Fault Correction	3	2	0	0	1	1	2	2

needed to locate and correct faults, with no reduction in effectiveness. However, these results do not tell us *why* these effects were achieved. To attempt to answer this question, we analyzed the screen recordings made during the user study.

4.7.1 Unreduced Recordings

On viewing the screen recordings obtained during the study, we noticed that none of the participants provided with unreduced recordings appeared to have a clear idea, initially, about the input fields or the lines of code responsible for the failure. They focused on the structure of the code as a whole and used that as a starting point for locating the fault and fixing it. We were able to enumerate three overall patterns adopted by participants for tackling unreduced recordings.

Approach 1: Undirected Search. Several participants, given unreduced recordings, spent most of their time scrolling through the code looking for the code responsible for the fault. Some participants spent all available time doing this, never actually locating the fault. For instance, one participant given an unreduced recording of the tic tac toe game (Application A), that was unresponsive when the user clicks on squares within the user interface, could not locate the code related to this unresponsiveness. Ultimately, he ran out of time.

Some participants combined the scrolling process with Google searches. They would scroll up and down considering code, and each time they suspected that a portion of code might be incorrect they would perform a Google search to verify their speculation. Provided with unreduced recordings, however, all such participants conducted Google searches related to non-faulty code. For instance, Application B contained a failure related to an inappropriate dialog box. Faced with this failure, one participant using an unreduced recording performed Google searches that were related to other non-faulty and fully functional dialog boxes.

Approach 2: Code-Understanding-Based Search. A second group of participants adopted an approach that consisted of trying to understand the code as a whole and determining its structure prior to locating the fault. First, they attempted to partition the code into a set of blocks differentiating different roles. One common behavior involved successively selecting specific areas of the code with the mouse. Other participants divided the code into blocks by adding new lines before and after each block in order to differentiate it from the rest of the code. Still others indented blocks of code in a certain manner and some inserted comments clearly separating the code into distinct parts.

We further divide these participants into those who, after partitioning code in some fashion, used one of two code-understanding processes: (a) incremental search and (b) directed search. The incremental search approach involves using Google searches to verify that each block of code is correct. Once the participants encountered a block of code that they believed was incorrect they focused on that block and attempted to correct it. For instance, Application C involves a company arranging pickup information at the airport for a hotel's clients. The failure is that the user can submit a request even if a phone number is input in the email text field. One participant partitioned the code for this application into three distinct parts, HTML, CSS and JavaScript. Then, he inserted comments in order to differentiate each block of code from the others. He further subdivided the JavaScript portion of the code into three parts, namely, catching the exception, throwing it and sending a confirmation message to the user. The participant conducted Google searches to ensure that the HTML and CSS blocks of code satisfied syntax requirements. He did not detect any inconsistencies in those blocks, so he focused his attention on the JavaScript portion of the code and conducted Google searches about the JavaScript that is responsible for catching errors. Again, the participant did not detect any inconsistencies, so he conducted Google searches about the second subpart of the JavaScript block that plays the role of throwing errors. After checking the rules for catching and throwing errors, the participant judged that one error was not caught appropriately, which he correctly concluded was the fault.

In contrast, the directed search approach consists of focusing the search effort directly on one specific block of code, considering each line. For example, Application D includes a form for renting cars, in which one button is unresponsive to the user's mouse clicks. After one participant replayed the recording, he examined the code and divided it into three parts: input manipulation, button clicks and error handling. The participant designated the button click portion of the code as being faulty and focused his attention on that block of code, ignoring other parts of the code. All the Google searches made by the participant were related to the block of code designated as faulty. After performing one search over one line of the faulty block, the participant indicated (correctly) that line as being faulty and attempted to correct the fault.

Approach 3: Exhaustive Inspection. A final group of participants adopted a different approach with unreduced recordings. Instead of directing their attention towards specific parts of the code, they inspected each line of code individually to attempt to judge its correctness. These participants performed many Google searches regarding lines of code, most of which were completely unrelated to the fault under consideration. For instance, on Application B (the application in which the failure involved an inappropriate dialog box output after a user clicked on the "Submit" button), one participant inspected every line of code even though some were completely unrelated to the submit button behavior. Furthermore, the participant conducted a Google search about each line of code and ended up running out of time.

4.7.2 Reduced Recordings

When using reduced recordings, participants appeared to have much clearer notions about which region of code to focus on. They did not spend much time scrolling through the code and viewing its structure; instead, they directed their attention towards a specific area of the code and worked on that. Some participants also retained keywords after viewing the reduced recordings and performed code searches in order to locate lines embedding those keywords – an approach not observed for participants given unreduced recordings. Also, participants used Google searches that were more directly focused on the occurrence of the fault and the failure.

For instance, consider Application A, the tic tac toe game that was unresponsive to user clicks on squares. One participant who was given a reduced recording revealing the associated failure directly focused on the area of code embedding the fault. The participant concluded that the unresponsiveness of clicks was caused by an incorrect calculation at the code level. Also, since the incorrect calculation was related to the variable "Square", the participant further narrowed his field of vision by searching the code for the keyword "Square". This participant was able to retain the name of the input field whose manipulation seemed to be causing the failure, then searched for that keyword at the level of the application.

Overall, it appears that, as postulated in the first paragraph of Section 2.1, participants who were given reduced recordings were

Table 5: Study 3 Objects

App	Downloads	User Rating	Size (LOC)	Language	Description and functionalities
Faq Forge [88]	17,716	4.0	1206	PHP, CSS, SQL, JavaScript	**Document Management** Create FAQS, manual guides and HOWTOs in a hierarchical structure
Time Clock [89]	16,733	5.0	6787	PHP, SQL, JavaScript	**Time Management** -Track employee schedule-Track upcoming vacations -Manage sign in sheets
School Mate [90]	77,158	4.6	19418	PHP, CSS, JavaScript, SQL	**School Management** -Admin: Manage classes/users -Teachers: Manage grades/assignments -Students: Access grades/submit homework -Parent: Check student progress

better able to focus on code and relevant keywords than those given unreduced recordings. It further appears that this is primarily due to the reduction in extraneous information provided by the reduced recordings, and the more easily observed connections between inputs that lead to failures and those failures themselves.

5. STUDY 3: SCALABILITY

Our third study considers the following research question:

RQ3: To what extent is Delta Debugging scalable to large and complex web applications?

5.1 Objects of Analysis

As objects of study, we chose three non-trivial applications that had been utilized in prior studies of techniques for testing web applications [4,56]. Table 5 provides details on the applications, illustrating the numbers of time they have been downloaded, their user ratings (on a scale of 1 to 5), their sizes in non-comment lines of code (measured using CLOC-Count Lines of Code),[6] the languages they utilize, and the functionalities they provide. The downloads and user ratings attest to the popularity of the objects.

We did not have access to actual faults for the web applications we selected, so we asked an experienced web application programmer who was not involved in this work, and had no information about our intended study, to place faults in each of the web applications that were representative of actual faults he had found in practice, and that resulted in detectable failures in the applications. During the fault seeding process, this programmer did uncover one actual fault in the first of the programs, `Faq Forge`. This seeding process resulted in the provision of six faults for `Faq Forge`, five faults for `School Mate`, and three faults for `Time Clock`. Details on the specific faults are shown in Table 6.

5.2 Variables

Our independent and dependent variables, and our process for obtaining values for them, are the same as those reported in Section 4.2 for Study 1.

5.3 Threats to Validity

The threats to validity for this study are also similar to those for Study 1, with the added external threat to validity posed by our use of seeded faults. However, our use of much larger and more complex web applications does help address the threat to validity posed by our use of smaller applications in the earlier studies.

5.4 Results and Analysis

Table 7 shows the differences in size and replay time observed for each recording before and after reduction. As the data shows, reduced recordings were all less than 28% the size of full recordings. The average size of full recordings was 48.4 elements while the average size of reduced recordings was 6.9 elements (overall, 14.0% the size of full recordings). The average replay time for full recordings was 35.6 seconds, while the average replay time for reduced recordings was 7.9 seconds (overall, 22.2% smaller than for full recordings). The table also shows the total amount of time required by our implementation of Delta Debugging itself; this time ranged from 31 seconds to 1,483 seconds (about 25 minutes), with an average of 461.8 seconds (about eight minutes).

[6]http://cloc.sourceforge.net/

Table 6: Study 3 Faults

App	Fault	Description
Faq Forge	1	Adding a 2nd page to an existing one causes both pages to display the content from the 2nd page
	2	From an existing topic with 3 pages, deleting page #2 and adding a new page #1 corrupts the page numbering currently displayed: Page#1 is labelled Page#3.
	3	Deleting a topic does not result in an actual deletion although the user receives a message confirming the deletion
	4	Creating a 3rd level child page is correctly shown on the current page but results into a file duplication in main
	5	Creating a document followed by clicking on the view document link results into showing the page number as -1 of 1.
School Mate	1	Dysfunctional "Change student association" functionality
	2	The page number is not displayed on one of the web pages
	3	The page generates an error when adding a new term
	4	The student cannot register for a class although there are seats available
	5	An announcement is not deleted even if the user clicks on "Delete"
Time Clock	1	The "Hours Worked" report only provides hours rounded down without any decimal portion
	2	The note entered on login/logout/ is not displayed under the notes column on the page
	3	The date is not being displayed in the top right corner of the timeclock.php page

To determine whether Delta Debugging produced statistically significant reductions in size and replay time, we conducted two-tailed Mann-Whitney U-tests [24] at significance level $\alpha = 0.05$. Our null hypotheses were “Recording reduction does not reduce recording size” and “Recording reduction does not reduce replay time”. The tests indicated that the reductions in size and replay time were both statistically significant, with p-values of 4.981e-8 in both cases.

5.5 Discussion

The reductions in size and replay time observed in this study are actually larger than those observed in the first study on smaller applications. Reduced recordings do contain, on average, more elements than those used in the first study, particularly on `FaqForge`, but they still do eliminate large numbers of elements. Of course, we cannot claim without further study that this reduction will en-

Table 7: Recording Size and Replay Time Before and After Reduction for Large, Complex Applications

App	Fault	Total DD Time	Size (elements)			Replay Time (sec)		
			Full Rec.	Red. Rec.	Pct.	Full Rec.	Red Rec.	Pct.
Faq Forge	1	613	41	9	22	34	11	32
	2	1,135	79	12	15	62	14	23
	3	760	58	9	16	44	11	25
	4	1,483	47	13	28	36	10	28
	5	471	38	10	26	38	12	32
	6	101	28	5	18	22	9	36
School Mate	1	220	134	5	4	89	6	7
	2	47	36	1	3	23	3	13
	3	133	39	6	15	25	6	24
	4	429	42	6	14	37	5	14
	5	172	34	6	18	27	7	26
Time Clock	1	631	42	9	21	20	7	35
	2	239	30	5	17	20	7	35
	3	31	30	1	3	22	3	14

able programmers to detect and correct faults more quickly, but in view of our second study we are hopeful that it will. The time required to run Delta Debugging itself is likely inflated by the fact that our implementation is a prototype, and we have not focused on performance in creating it; however, the execution is fully automated, and if it does enable developers to detect and correct faults more efficiently and effectively, it is most likely worth it.

6. RELATED WORK

Many techniques (e.g., [6,13,23,50]) attempt to detect failures at runtime using static or dynamic analyses. Many techniques rely on testing-based approaches to detect failures, and several such techniques have been explored in relation to web applications (e.g., [31, 33–36]). There has also been work on approaches for reproducing failures (e.g., [16, 18]). Such techniques focus on detecting and reproducing failures, whereas in this work we focus on helping debuggers localize faults from failures that are already known.

Numerous approaches attempt to help developers localize faults in non-web-based applications, using a wide variety of tactics (e.g., [1, 15, 17, 25, 27, 29, 43, 45, 54]). Recent work has also considered fault localization for web application software. For example, Ocariza et al. [39] provide an automated method for localizing DOM-related JavaScript faults through dynamic analysis, and tracing and slicing of JavaScript code, and Artzi et al. [3] provide a technique for performing fault localization on dynamic PHP web applications. The primary difference between these approaches and the approach we present here involves our focus on behavioral recordings and recording reduction.

There has also been work on facilitating the debugging process from the human standpoint. Whyline [20] facilitates debugging by presenting the programmer with an interface that allows them to formulate questions about a program's behavior. Vejovis [40] helps debuggers by suggesting fixes for DOM-related JavaScript faults. Our work shares their goal of helping debuggers in their tasks.

Strategies closely related to ours are those that attempt to minimize program input, program state, or test cases that produce failures. Regehr et al. [44] use test case reduction to detect C compiler bugs, and Lei and Andrews [26] minimize randomly generated failing tests. Most closely related to this work, given that we draw on its algorithm, is Delta Debugging, which has been utilized in several different scenarios. Zeller and Hildebrandt [53] use Delta Debugging to simplify and isolate input that reproduces failures. Zeller [52] also adapts Delta Debugging to narrow down failure causes involving program state. Cleve and Zeller [11] use Delta Debugging to create minimal test cases that produce failures. Burger and Zeller [8] minimize Java programs focusing on object interactions. None of these approaches considers web applications, or involves recording, replaying, and reducing recordings at the level of program behavior that we consider; in this work we show how Delta Debugging can be used to reduce recordings of web applications.

There has been considerable research on record-replay approaches. Many such approaches have been explored in the context of non-web software (e.g., [9,19,37,38,41,49]), without considering recording reduction or targeting web applications. Clause and Orso [10] present a technique for recording and replaying program executions that does perform reduction, with the aim of helping replicate and correct code that fails in the field. This technique focuses on compiled code in non-web applications, and on tracking interactions in the code by intercepting various I/O streams. In contrast, we focus on user interactions with web applications, captured by tools that monitor information related to events and DOM trees.

There has been some research on record/replay infrastructures for web-based software. Sen et al. [48] present Jalangi, a program analysis framework for JavaScript that incorporates record/replay. Andrica and Candea [2] present WaRR, a tool that records and replays the interaction between users and web applications. Mickens et al. [32] present Mugshot, which allows deterministic replay of JavaScript programs. Burg et al. [7] present Dolos, a deterministic record/replay infrastructure for web applications that captures not only user inputs but also network callbacks. During playback, captured inputs are re-delivered to the browser engine but new "live" inputs are suppressed. Asynchronous events such as animation timers and futures are captured and replayed using hooks inside the browser engine. Return values of nondeterministic functions that provide JavaScript with access to persistent state (browser cookies, local storage, etc.) and environmental data (current time, screen size, etc.) are memoized. None of this work, however, has considered recording reduction.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach for recording reduction based on Delta Debugging that operates on recordings of web applications. We presented empirical evidence that our approach can significantly reduce the size and replay time of recordings, that reduced recordings help programmers debug more cost-effectively, and that the approach itself can scale to larger, more complex applications.

Given the results of our studies, we intend to next explore the application of the approach in conjunction with deterministic record/replay infrastructures and with alternative oracle types. We also intend to study reduction algorithms that analyze dependencies or incorporate such analyses into the Delta Debugging approach. Finally, we plan to extend the scope of our empirical studies.

8. ACKNOWLEDGEMENTS

This work has been partially supported by the National Science Foundation through award IIS-1314365. We thank Wayne Motycka for his assistance with study infrastructure and data analysis, and we thank Cyrille Nzouda of the Department of Statistics at the University of Nebraska - Lincoln for assistance with statistical approaches. We also thank our study participants.

9. REFERENCES

[1] M. A. Alipour and A. Groce. Extended program invariants: Applications in testing and fault localization. In *WODA*, pages 7–11, 2012.

[2] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, pages 403–410, June 2011.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *ICSE*, pages 265–274, 2010.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA*, pages 261–272, July 2008.

[5] K. Arya, T. Denniston, A.-M. Visan, and G. Cooperman. Semi-automated debugging via binary search through a process lifetime. In *PLOS*, November 2013.

[6] Y. Brun. Software fault identification via dynamic analysis and machine learning. Master's thesis, Massachusetts Institute of Technology, August 2003.

[7] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, pages 473–484, 2013.

[8] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *ISSTA*, pages 221–231, July 2011.

[9] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible debugging using program instrumentation. *TSE*, 27(8):715–727, August 2001.

[10] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE*, pages 261–270, May 2007.

[11] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *WAD*, August 2000.

[12] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. de Bosschere. A taxonomy of execution replay systems. In *MTI*, 2003.

[13] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17(2):1–37, April 2008.

[14] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *OSP*, Oct 2009.

[15] Peifeng H., Zhenyu Z., W. K. Chan, and T. H. Tse. Fault localization with non-parametric program behavior model. In *CQS*, pages 385–395, Aug 2008.

[16] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. *SIGPLAN Notices*, 48(6):141–152, June 2013.

[17] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[18] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Reproducing field failures for programs with complex grammar-based input. In *ICST*, pages 163–172, March 2014.

[19] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, pages 1–1, 2005.

[20] A. J. Ko and B. A. Myers. Chi. In *CHI*, pages 151–158, 2004.

[21] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about Java program output. *TOSEM*, 20(2):1–34, August 2010.

[22] A. J. Ko and X. Zhang. FeedLack detects missing feedback in web applications. In *CHI*, 2011.

[23] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee. Using assertions to help end-user programmers create dependable web macros. In *FSE*, pages 124–134, 2008.

[24] O. Koresteleva. *Nonparametric Methods in Statistics with SAS Applications*. CRC Press, Boca Raton, FL, 2004.

[25] F.-C. Kuo, T. Y. Chen, H. Liu, and W. K. Chan. Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters. *SQC*, 16(3):303–327, September 2008.

[26] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE*, pages 267–276, 2005.

[27] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE*, pages 417–420, November 2007.

[28] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *HFCS*, pages 1719–1728, 2008.

[29] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, June 2005.

[30] A. Marron, G. Weiss, and G. Wiener. A decentralized approach for programming interactive applications with JavaScript and blockly? In *PSLA*, pages 59–70, 2012.

[31] A Mesbah, A van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *TSE*, 38(1):35–53, Jan 2012.

[32] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *USENIX*, 2010.

[33] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *ASE*, 2014.

[34] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *WE*, pages 238–252, 2012.

[35] S. Mirshokraie, A Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *ICST*, pages 74–83, March 2013.

[36] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. PYTHIA: Generating test cases with oracles for JavaScript applications. In *ASE*, pages 610–615, 2013.

[37] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *CAN*, 33(2):284–295, May 2005.

[38] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI*, pages 313–325, 1994.

[39] F. Ocariza, K. Pattabiraman, and A Mesbah. AutoFLox: An automatic fault localizer for client-side JavaScript. In *ICST*, pages 31–40, April 2012.

[40] F. Ocariza, K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *ICSE*, 2014.

[41] A. Orso and B. Kennedy. Selective capture and replay of program executions. *SEN*, 30(4):1–7, May 2005.

[42] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, July 2011.

[43] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss.

Automated fault localization using potential invariants. In *AAD*, pages 273–276, September 8–10, 2003.

[44] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. *SIGPLAN Notices*, 47(6):335–346, June 2012.

[45] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[46] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, pages 677–694, 2011.

[47] The Selenium Project, Selenium WebDriverDocumentation, 2012. http://seleniumhq.org/docs/03_webdriver.html/.

[48] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *FSE*, pages 488–498, 2013.

[49] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.

[50] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA*, pages 97–107, 2007.

[51] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: Using GUI screenshots for search and automation. In *UIST*, pages 183–192, 2009.

[52] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, October 2002.

[53] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2):183–200, February 2002.

[54] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *FSE*, pages 43–52, September 2009.

[55] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C Weiss. What makes a good bug report? *TSE*, 36(5):618–643, 2010.

[56] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao. Virtual DOM coverage for effective testing of dynamic web applications. In *ISSTA*, pages 60–70, July 2014.

[57] http://stackoverflow.com/questions/3101402/the-travel-tickets-problem.

[58] http://stackoverflow.com/questions/16266704/how-can-i-integrate-javascript-onclick-function-with-php.

[59] http://stackoverflow.com/questions/25037819/watin-submit-button-remain-disabled-after-filling-form.

[60] http://stackoverflow.com/questions/8487506/how-to-structure-a-click-to-win-type-comptetition-with-a-set-amount-of-random.

[61] http://stackoverflow.com/questions/16810769/how-to-make-datepicker-disabled-on-public-holiday-disabled-on-sunday-and-disabl.

[62] http://stackoverflow.com/questions/24284859/jquery-selectors.

[63] http://stackoverflow.com/questions/17445722/javascript-based-on-value-in-one-drop-down-bow-changes-values-in-2-other-drop-do.

[64] http://stackoverflow.com/questions/23225556/add-new-recipe-with-ajax-pop-up-for-ingredients.

[65] http://stackoverflow.com/questions/22841981/car-loan-calculator-in-javascript-displays-nothing.

[66] http://stackoverflow.com/questions/15855790/javascript-validation-is-not-working-in-html-registration-form.

[67] http://stackoverflow.com/questions/2157361/how-to-get-a-select-options-list-if-i-dont-have-a-form-in-the-dom.

[68] http://stackoverflow.com/questions/14297463/recognize-visitors-with-javascript-cookie-for-survey.

[69] http://stackoverflow.com/questions/22194655/jquery-code-is-not-working.

[70] http://stackoverflow.com/questions/11895387/monkey-with-rental-cars-integration-into-third-party-websites.

[71] http://stackoverflow.com/questions/597769/how-do-i-create-an-abstract-base-class-in-javascript/597872#597872.

[72] http://stackoverflow.com/questions/20726565/dynamic-dropdown-select-option-first-option-from-the-service-and-the-second-opti.

[73] http://stackoverflow.com/questions/18432116/student-average-no-response-in-javascript.

[74] https://stackoverflow.com/questions/15166552/javascript-lottery-game-using-random-numbers-what-is-wrong-with-my-logic.

[75] http://stackoverflow.com/questions/26101573/editing-drop-down-options-based-on-previous-drop-down-options.

[76] http://stackoverflow.com/questions/8350551/javascript-checking-if-form-is-null.

[77] http://stackoverflow.com/questions/25056856/getting-a-javascript-validation-error-on-signup-page-when-clicking-create-accou.

[78] http://stackoverflow.com/questions/15508772/using-jquery-to-hide-show-links-based-on-checkbox.

[79] http://stackoverflow.com/questions/28096735/how-do-i-access-passports-req-user-variable-in-client-side-javascript.

[80] http://stackoverflow.com/questions/20436140/javascript-shopping-cart-calculator.

[81] http://stackoverflow.com/questions/15152494/tic-tac-toe-with-javascript.

[82] http://stackoverflow.com/questions/9543203/javascript-form-validation-for-user-account.

[83] http://stackoverflow.com/questions/12434066/calculate-age-javascript-in-adobe-acrobat.

[84] http://stackoverflow.com/questions/2841898/how-to-validate-username-using-javascript.

[85] http://stackoverflow.com/questions/17122516/writing-numbers-1-80-in-javascript-and-writing-text-for-certain-multiples.

[86] http://stackoverflow.com/questions/5939474/php-add-to-shopping-cart-problem.

[87] http://stackoverflow.com/questions/8269445/javascript-drag-and-drop-removing-dragged-element-following-successful-drop.

[88] http://sourceforge.net/projects/faqforge/.

[89] https://sourceforge.net/projects/schoolmate/.

[90] https://sourceforge.net/projects/timeclock/.