# Dicer: A Framework for Controlled, Large-Scale Web Experiments

Sarah Chasins
University of California, Berkeley, USA
schasins@cs.berkeley.edu

Phitchaya Mangpo Phothilimthana
University of California, Berkeley, USA
mangpo@cs.berkeley.edu

## ABSTRACT

As dynamic, complex, and non-deterministic webpages proliferate, running controlled web experiments on live webpages is becoming increasingly difficult. To compare algorithms that take webpages as inputs, an experimenter must worry about ever-changing webpages, and also about scalability. Because webpage contents are constantly changing, experimenters must intervene to hold webpages constant, in order to guarantee a fair comparison between algorithms. Because webpages are increasingly customized and diverse, experimenters must test web algorithms over thousands of webpages, and thus need to implement their experiments efficiently. Unfortunately, no existing testing frameworks have been designed for this type of experiment.

We introduce Dicer, a framework for running large-scale controlled experiments on live webpages. Dicer's programming model allows experimenters to easily 1) control when to enforce a *same-page guarantee* and 2) parallelize test execution. The *same-page guarantee* ensures that all loads of a given URL produce the same response. The framework utilizes a specialized caching proxy server to enforce this guarantee. We evaluate Dicer on a dataset of 1,000 real webpages, and find it upholds the same-page guarantee with little overhead.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## Keywords

Web Algorithm Testing, Testing Framework, JavaScript

## 1. INTRODUCTION

Algorithms that take webpages as inputs are becoming increasingly ubiquitous. From search algorithms that find data on the web, to template extraction algorithms that identify data relations on the web, to scraping algorithms that collect data from the web, the number of programs that operate over web content is on the rise. As the amount of online data continues to increase, the need for web algorithms — that is, algorithms that take webpages as inputs — will only grow.

Unfortunately, support for the development of web algorithms has not kept pace. In particular, *controlled* testing remains extremely difficult. While there is a preponderance of tools targeted at developers testing their own pages, these tools are not easily applied to the more general problem of running arbitrary tests over real-world, constantly changing pages. If we are to fairly compare different algorithms on real-world pages, we cannot simply run the algorithms directly; input webpages can be updated at any time, perhaps altering an experiment's outcome, making one algorithm or another appear more successful. In *controlled* testing, an experimenter must hold input webpages constant while comparing different algorithms. To our knowledge, no existing JavaScript testing framework offers this functionality.

As a motivating example, we discuss one such web algorithm, the DOM node addressing problem. The task is to load a URL at time $t_1$, describe a given node $n$ from the loaded webpage, then load the same URL at a later time $t_2$, and use the description of $n$ to identify the node at time $t_2$ that corresponds to $n$. Several node addressing algorithms have been proposed [1, 2, 6, 11, 18], but because testing them is so difficult, none has been evaluated empirically.

In response to the lack of testing tools for web algorithms, we introduce Dicer, the DOM-Interacting Controlled Experiment Runner. Dicer uses a custom caching proxy server to offer a *same-page guarantee*, a guarantee that all requests for a given URL return the same response, regardless of server state changes and JavaScript non-determinism. Further, Dicer automatically parallelizes experiments, facilitating large-scale evaluations. Our goal is to make web experiments more accessible, and thereby make thorough testing of web algorithms more widespread. Dicer's programming model makes it easy for users to reason about input pages and hold them stable over time, and it ensures that automatic parallelization is always possible. This brings controlled web experiments within reach of a wider audience.

In summary, we make the following contributions:

- We develop a novel programming model for running DOM-interacting controlled web experiments.
- We design and implement a proxy server that offers a stricter same-page guarantee than existing proxy servers.
- We design and implement the first JavaScript testing tool that can run DOM-interacting controlled web experiments.

## 2. GOALS

We identify five core properties that are critical to making a framework capable of running large-scale controlled experiments on real-world webpages. A framework must:


*WWW 2015 Companion,* May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2741699 .

1. parallelize test execution
2. allow DOM interaction
3. run arbitrary JavaScript code
4. run on live pages from URLs (not only local DOMs)
5. offer a *same-page guarantee*

We address the need for each in turn. First, because our focus is on large-scale experiments, parallelization is key to making our target experiments practical. Second, because we target algorithms that take webpages as inputs, it is crucial to allow algorithms to use the DOM API to access and interact with webpage content. Tools that only test JavaScript functions independently of webpages, such as Node.js testing frameworks, are not suitable for our target domain. Third, a framework should permit the user to test arbitrary JavaScript code. A framework that only allows pass/fail outputs restricts the class of algorithms that can be tested. Fourth, it is important that a framework allow users to run their tests on real pages, not only on locally constructed DOMs. While users can of course download pages from URLs in order to construct their DOMs locally, this process becomes cumbersome when a user wishes to test at multiple different points in time. Last, comparing different web algorithms demands a same-page guarantee. If a user wishes to test one DOM-interacting algorithm against another, it is crucial that both receive the same page as input.

## 2.1 Motivating Example

To offer robust record and replay for live webpages, a tool must be able to identify the same DOM node in a webpage over time, even if the DOM structure of the page changes between record and replay. We will call this the *node addressing* problem. Using an XPath from the DOM tree's root to the target node is too fragile; new wrapper nodes and sibling nodes will break the XPath. Ids would be sufficient if all elements had ids, if those ids stayed the same throughout page redesigns, and if all web designers adhered to the one element per id rule. Unfortunately, none of these conditions is met across the web. Handling real webpages requires much more care, and node addressing algorithms are typically quite complex.

We formalize a node addressing algorithm as a function from a DOM tree $T$ and a node $n \in T$ to an expression $e$, such that $e(T) = n$. Let $T'$ be a different DOM tree, and let $e(T') = n'$. We consider a node addressing algorithm robust if $n'$ is the same node that a human would select if asked to find the original node $n$ in $T'$.

Node addressing algorithms are in use in tools like CoScripter [11], iMacros [1], and Selenium's Record and Playback [18]. Unfortunately, testing node addressing algorithms is so difficult that to our knowledge, despite the fact that this problem is central to the success of their tools, none of these node addressing algorithms have been tested against other candidate algorithms on large data sets. Each project appears to have settled on an approach that works sufficiently well on its test cases, without empirical validation.

Our motivating example will be an experiment that tests multiple node addressing algorithms against each other. Since we will not ask a human to evaluate $n'$ in our large-scale experiment, we use a simple correctness condition. If algorithm A produces $e$ such that $e(T) = n$ and $e(T') = n'$, we will consider algorithm A correct on page $T'$ if and only if clicking on $n'$ directs the browser to the same URL as clicking on $n$. Since many nodes will not respond to clicking, we only test on nodes that direct the browser to a new URL. We term these *reactive* nodes.

| Method | Description |
|---|---|
| startSession() | Starts a new session. |
| endSession() | Ends current session. |
| stage(String ip, String it, String ot) | Adds stage to the current session with input program from file *ip*, input table from file *it*, which will write to file *ot*. |

Table 1: The Dicer API.

Note that an experiment like this relies on all five of the crucial features identified above.

1. Parallelism: A thorough test demands running this task on many nodes, in order to reveal enough naturalistically broken addresses to distinguish between approaches, which makes parallel execution highly desirable.
2. DOM interaction: The algorithms must click on nodes.
3. Arbitrary JavaScript: The algorithms cannot be limited to, for instance, pass/fail outputs.
4. Running on live pages: The algorithms should run on the real sites of interest. The pages should change between training and testing.
5. Same-page guarantee: To fairly compare different algorithms, their inputs must be the same.

# 3. PROGRAMMING MODEL

In this section we introduce the core abstractions that form Dicer's programming model, describe how our motivating example is implemented with these abstractions, and discuss some design decisions.

## 3.1 Abstractions

Our programming model is built around a few key abstractions that make it easy to express complicated experiments.

**Session**: a sequence of **stages**.
**Stage**: a (**input table**, **input program**) pair, which produces an **output table** when passed to Dicer.
**Input Table**: a set of $n$-field rows.
**Input Program**: a sequence of one or more **algorithms**.
**Algorithm**: a set of one or more **subalgorithms**.
**Subalgorithm**: a JavaScript function, accepts $n$ arguments.
**Output Table**: a set of $m$-field rows.

A session is a sequence of stages during which Dicer offers a same-page guarantee. During a session, loading a given URL always loads the same DOM tree. Each stage is defined by its input table and its input program. The first column of the input table contains URLs, indicating the pages on which the input program should be run. All remaining columns contain additional arguments to the input program. An input program may contain multiple algorithms to run for each row in the input table. If an algorithm runs over multiple pages, it must be decomposed into subalgorithms, one to run on each page. Each subalgorithm is a function that accepts $n$ arguments (corresponding to the $n$ items in each input table row). Each row in the input table corresponds to one run of the input program. For each algorithm in the input program, Dicer first directs the browser to the URL in the first column of the row, then runs the algorithm on the loaded page, passing all row cells as arguments to the algorithm. Thus, for a given row, all algorithms are run with the same arguments on fresh, identical copies of the target page. For any given input row, the return values of the different algorithms are concatenated to produce full output rows, which are appended to the output table. Only the final subalgorithm of an algorithm may produce output.

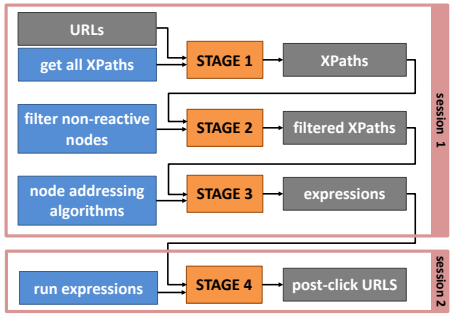

Figure 1: The motivating example. Blue boxes represent JavaScript programs, and gray boxes represent tables.

Dicer offers a simple API for interacting with these abstractions, outlined in Table 1.

## 3.2 Motivating Example Experiment

To explain the Dicer programming model more concretely, we show how we can use the framework to implement the node addressing experiment described in Section 2.1. In Dicer, this experiment is split into the four stages depicted in Figure 1 and described below.

**Stage 1** traverses the DOM tree, recording an XPath for each node. The input table has only one column, the URL column. The table contains the list of URLs on which we want to test our node addressing algorithms. The input program is an algorithm that produces an XPath for each node in a document's DOM tree, and emits an output row for each XPath.

**Stage 2** determines which nodes are reactive. Its input table is the output table from Stage 1. For each XPath in the input table, it clicks on the node at the XPath. (Since Stages 1 and 2 run in a single session, with the same-page guarantee ensuring the same $T$, even a fragile XPath will serve here.) The input program is one algorithm with two subalgorithms. The first subalgorithm clicks on the node identified by the XPath. If the click loads a new page, the second subalgorithm runs on the new page. If the click does not load a new page, the second subalgorithm runs on the original page. By retrieving the current URL in the second subalgorithm and comparing it with the pre-click URL, we can determine whether the clicked node is reactive [1]. The subalgorithm only produces an output row if the URL has indeed changed. For this stage, the output table again contains one row per XPath, but now all rows that correspond to non-reactive nodes have been removed.

**Stage 3** takes the XPaths of all reactive nodes as input, and runs each node addressing algorithm on each reactive node, producing $e$ expressions as output. The input table has one row for each reactive node. The first column contains the URLs at which those reactive nodes are found. The second column has the XPaths of the nodes. The input program is a set of algorithms, all the node addressing algorithms we want to test. Each of these algorithms has only one subalgorithm, a JavaScript function that takes all input columns, including the XPath, as its arguments. Each algorithm returns an expression $e$. All the algorithms' outputs together form an output row. Thus the output table contains a column corresponding to each node addressing algorithm.

[1]Redirects and JavaScript-controlled target URLs make clicking the only reliable way to collect post-click URLs.

Listing 1: Using the Dicer API to configure Dicer to run our motivating example.

```
d = new Dicer(); // New framework instance.
d.startSession();
// Here urls.csv is a list of pages on which to run.
d.stage("collectXPaths.js","urls.csv","xPaths.csv");
// Below, the user strings together stages, using
// the results of stage 1 as the input for stage 2.
d.stage("filterXPaths.js","xPaths.csv","filteredXPaths.csv");
d.stage("saveNodes.js","filteredXPaths.csv","expressions.csv");
d.endSession();
d.startSession();
d.stage("retrieveNodes.js","expressions.csv","results.csv");
d.endSession();
```

**Stage 4** tests the $e$ expressions, so it takes the $e$ expressions as input. If we were to test the algorithms' outputs — the $e$ expressions — in the same session, our results would be rather boring. If $T' = T$, all algorithms should succeed on $T'$. Thus, we test the algorithms' outputs in a new session, so that pages are allowed to change. This stage runs one algorithm per node addressing algorithm. Each algorithm uses an $e$ expression to find a node on the new page, and then click on it. It produces the new post-click URL as output. The new post-click URL can be compared with the Stage 2 post-click URL to determine whether each algorithm successfully identifies the corresponding node.

To run this experiment with Dicer, a user only needs the JavaScript input programs described above, a list of URLs on which to run the experiment, and the code in Listing 1, which uses the Dicer API to set up the experiment.

## 3.3 The Design of Stage Output

Recall that a single input row may produce any number of output rows. Also recall that we allow subalgorithms. Together these features have the potential to complicate our programming model. We keep the design simple and usable by allowing only the final subalgorithm to produce output.

An alternative approach would allow all subalgorithms to produce output. This is useful for cases in which earlier subalgorithms can access data that later subalgorithms cannot, but it substantially complicates the programming model, because the outputs of different subalgorithms may vary in number, and must somehow be stitched together to produce complete output rows. Fortunately, algorithms that would benefit from the more complicated programming model can be refactored to adhere to our simpler one. For instance, recall that our motivating example needs to compare the URLs before and after clicking on a node. The pre-click URL is available during the first subalgorithm, but not during the later subalgorithm. To complete this task in the simple programming model, it is split into two stages. The first stage stores the original URL, while the second clicks the link and stores the second URL. Because this approach simplifies the user's experiment design experience, and because splitting such tasks across stages is sufficient to make our approach general, this is the design we have adopted.

We considered allowing all subalgorithms to produce output, but requiring the user's JavaScript subalgorithm code to associate each slice of an output row with a row ID. All cells in a row would share an ID. We believe that this requirement complicates the programming model, putting a burden even on users with simple tests. We rejected this approach, choosing to optimize for usability in the common case. Another alternative design would require that the number of output rows be the same across subalgorithms, allowing the

Figure 2: Different algorithms may return different numbers of rows for a given input row. Here algorithms $a$, $b$, and $c$ produce 2, 1, and 3 outputs respectively. They are stitched together to create output rows 1, 2, and 3.

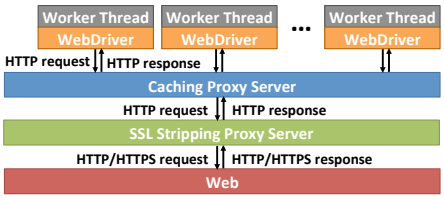

Figure 3: Dicer architecture. Dicer runs multiple workers in parallel. Each worker owns a web driver — a browser instance that can be controlled programmatically. The network traffic goes through a caching proxy server.

framework to use ordering to align slices and produce the full rows. A final option would allow each algorithm to produce only one output row per input row. We decided against these last two designs because of how substantially they diminish the expressiveness of the model.

Although only one subalgorithm produces output, all algorithms must be allowed to produce as many output rows as desired, even if different algorithms produce different numbers of output rows for the same input row. Output slices are stitched together based on the order in which they are returned by the algorithms, as shown in Figure 2. This approach simplifies users' code for the common cases in which there is only a single algorithm, each algorithm returns only a single row, or each algorithm returns the same number of rows. However, it restricts the model's expressiveness in one scenario; if algorithms do not know the order in which other algorithms will return their output rows, but still want to achieve a particular alignment with each other, this design will not be convenient. Experimenters with this use case would be best served by an ID alignment approach like the one described above for subalgorithms. Because of the burden it places on simple experiments, we did not choose an ID alignment model. However, advanced users can leverage order alignment to simulate ID alignment by associating each algorithm output with an ID, then emitting outputs in an ID-defined order. Thus our design optimizes for usability in the common case, but provides users enough flexibility to handle the uncommon alignment case.

# 4. IMPLEMENTATION

The basic architecture of Dicer, illustrated in Figure 3, revolves around directing a set of worker threads, each of which controls a browser instance. Their web traffic goes through our custom proxy server, which implements a caching scheme that upholds our same-page guarantee.

Source code for Dicer is available at `https://github.com/schasins/dicer`.

## 4.1 The Dicer Library

We have implemented our framework as a Java library. Users interact with the simple abstractions described in Section 3.1, as in Listing 1. Because Dicer is implemented as a Java library, users can interleave Dicer processing with other Java processing as necessary — for instance to combine the outputs of two stages into an input for another stage.

Dicer uses Selenium [3] to control browser instances. Headless WebKits were not sufficiently robust or reliable for our domain. Prototype implementations of Dicer built on top of PhantomJS [16] and Ghost.py [9] had respectively 0.6% and 12.4% rates of incorrect answers on a simple title extraction benchmark. In contrast, the Selenium implementation yields 0 incorrect answers on the same benchmark.

Dicer automatically parallelizes experiments. It runs a given input program across multiple input table rows in parallel. Also, Dicer can run multiple algorithms from a given input program across each input row in parallel. The Dicer programming model ensures that it is always correct to parallelize these components of an experiment. Dicer uses a shared queue to distribute tasks across worker threads.

## 4.2 The Dicer Proxy Server

Enforcing our same-page guarantee requires controlling for both non-determinism in the server and non-determinism in the pages' own JavaScript code.

### *4.2.1 Server-Level Non-Determinism*

Dicer uses a caching proxy server to enforce the same-page guarantee. It makes a new cache whenever a new session is created and serves all recurring URLs from that cache until the end of the session. All URLs, including URLs loaded directly by Dicer, URLs loaded within iframes, and URLs loaded via AJAX go through the proxy server.

Despite the preponderance of existing caching proxy servers, none proved sufficiently configurable to meet Dicer's needs. Squid cache [19, 21] can be configured to ignore some web cache policy parameters (e.g. 'no-cache,' 'must-revalidate,' and 'expiration'), but not others (e.g. 'Vary'). Thus, Squid will never cache any page with "Vary: *" in the header. Apache Traffic Server [5] provides even less cache configuration control. Other proxy servers can be configured to ignore all caching policy parameters, but are too fragile to be useful for large-scale experiments. Some can only handle HTTPS traffic with restrictions. For instance, Polipo [7] can be configured appropriately, but it is not very stable, and cannot decrypt or modify HTTPS traffic.

To meet our framework's demands, we implemented a custom caching proxy server. Dicer directs all request-response traffic through our caching proxy server, and the caching proxy server in turn directs all request-response traffic through an SSL-stripping proxy server [13], as illustrated in Figure 3. The SSL-stripping layer allows our proxy server to handle HTTPS traffic. It decrypts responses from the original servers and forwards the plain text to the caching proxy server, as if they are HTTP responses. Thus, the proxy server can freely modify the content of any response, even an HTTPS response. This is critical to mitigating JavaScript non-determinism, as discussed in Section 4.2.2.

Our custom proxy server stores every response into its cache, ignoring all web cache policy parameters in the header. When a framework browser instance requests a given URL, it always elicits the same response from the server. The only exception to the permanent caching rule addresses cyclic redirects. Some pages use a cyclic redirect process, redirecting a request for URL $X$ to URL $Y$ (with a'no-cache' policy), and redirecting a request for $Y$ to $X$, until eventually the originally requested $X$ is ready, and the $X$ response is no longer a redirect. At this point, the response for $X$ contains the final page content that our cache should as-

sociate with URL $X$. In this scenario, if neither $X$'s nor $Y$'s associated response can be altered, our system will loop forever. Our server addresses this issue by maintaining a redirect table and running a cycle check before recording a redirect response. If the addition of the redirect response would cause such a cycle, the server removes the pre-existing cached response that produces the cycle.

#### 4.2.2 JavaScript-Level Non-Determinism

Ensuring server-level determinism is not sufficient to offer a same-page guarantee. Even if a given URL always retrieves the same response, the page's own JavaScript is a source of non-determinism. Non-deterministic JavaScript may itself modify the page, or may make new requests to the server.

The Mugshot project [14] identifies `math.random` and the JavaScript `Date` class as the two JavaScript functions that introduce non-determinism. Since their work, Navigation Timing has been introduced and become prevalent, and is now another substantial source of JavaScript non-determinism. To prevent these sources of non-determinism from undermining our same-page guarantee, our proxy server inserts a script into all HTML responses. This script replaces `math.random` with a deterministic, explicitly seeded algorithm. During a given session, all pages receive the same seed. The script also replaces the `Date` constructor, which returns the current date, with a constructor that returns the date at the beginning of the current session. Finally, the script replaces `window.performance` with an empty dictionary to handle variations in Navigation Timing.

Cookies are also a potential source of JavaScript-level non-determinism, since they can change during an experiment. Thus Dicer turns off cookie storage for its browser instances.

Note that these are not the only sources of non-determinism in the browser. For instance, the browser's scheduler is non-deterministic. If multiple requests are outstanding at once, or if a page's JavaScript uses the `setTimeout` function, scheduler non-determinism may cause the page to diverge. However we find these ordering-based sources of non-determinism have little effect in practice. (See Section 5.1 for evaluation details.)

## 5. EVALUATION

In this section we evaluate how well Dicer upholds the same-page guarantee, as well as the effect of the same-page guarantee on framework execution time.

### 5.1 Same-Page Guarantee

To assess our framework's same-page guarantee, we use a data set of more than 200,000 DOM nodes — specifically, all nodes in the Alexa top 100 webpages. We load each webpage twice in a single Dicer session and compare DOMs. We observe the percentage of unmatched nodes with no non-determinism control in place, with only server-level non-determinism control in place, and with both server- and JavaScript-level non-determinism control in place. We consider a node matched if both its position in the DOM tree and its text content is the same across loads.

Table 2 displays our results. Note that the failure to sufficiently control for non-determinism may affect our results in two ways. It may mean that a page times out in one load but not another, or it may mean that a page is loaded both times, but with different content. In Table 2, the "Pre-Filtering" row reflects the percentages of unmatched nodes, including nodes that are considered to be unmatched because a page times out during only one of its loads. The "Post-Filtering" row reflects the percentage of unmatched nodes after the removal of nodes from pages that timed out in one of the loads. That is, the set of nodes is first restricted to the nodes from pages that successfully loaded both times. We then identify unmatched nodes in that restricted set.

| | No NDC | S NDC | S + JS NDC |
|---|---|---|---|
| Unmatched Pre-Filtering | 17.7% | 14.5% | 0.1% |
| Unmatched Post-Filtering | 7.4% | 6.1% | 0.1% |

Table 2: The percentage of unmatched nodes across two loads of the same pages when there is no non-determinism control (No NDC), when only server non-determinism is controlled (S NDC), and when both server and JavaScript non-determinism is controlled (S + JS NDC). The "Pre-Filtering" row includes nodes that go unmatched because of non-deterministic timeouts.

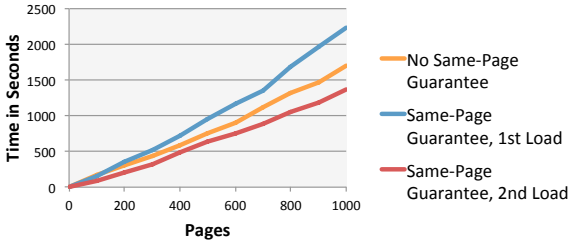


Figure 4: A comparison of the execution time in seconds for runs with no same-page guarantee, first loads with the same-page guarantee, and second loads with the same-page guarantee, by the number of pages loaded.

When we control for both server- and JavaScript-level non-determinism, as described in Section 4, only 0.1% of nodes were unmatched. Also note that when both server- and JavaScript-level non- determinism are controlled, non-deterministic timeouts are eliminated. We conclude that our approach to non-determinism control is sufficient for our target domain. While we could obtain higher determinism guarantees by building a custom browser with a deterministic scheduler, this approach would sacrifice the performance and external validity advantages of using a real production browser. We leave heavyweight techniques with stronger determinism guarantees for future work.

### 5.2 Performance Impact

To assess the overhead associated with upholding the same-page guarantee, we compare loading time with and without our caching proxy server in place. We consider both the first loading time, at which point the proxy server must retrieve and store all responses, and the second loading time, at which point it serves responses from the cache. We compare the execution times of a simple title extraction benchmark on Alexa's top 1,000 webpages [4].

Figure 4 reveals that the first load with the same-page guarantee exhibits a 31.1% slowdown over loads without the same-page guarantee. This is the overhead associated with modifying and saving the retrieved pages, a cost incurred for the first load of any URL. In contrast, the second load with the same-page guarantee exhibits a 19.4% speedup over loads without the same-page guarantee. This reflects the performance benefits of caching.

Since the performance impact on first loads is manageable, and the performance of all future loads is better with the same-page guarantee, we conclude that our framework's

execution times are well within the acceptable range. In light of the fact that our proxy server has not yet been optimized for performance, we are satisfied that the cost of upholding the same-page guarantee is low.

### 5.3 Motivating Example

We used Dicer to test five node addressing algorithms on a dataset of 25,000 nodes. The results determined the node addressing approach of a record and replay tool, Ringer [2].

## 6. RELATED WORK

Of the many existing tools for running JavaScript tests, almost all are targeted towards web developers who want to test their own pages, or even just their JavaScript. We discuss the main subcategories of this class of tools.

We start with tools targeted towards web developers running unit tests. Jasmine [8] is one of the most prevalent tools. While its ease of use makes it an excellent tool for small-scale experiments, it lacks many of the characteristics we desire for large-scale, general purpose web experiments. First, its parallelization mechanism is quite limited. Second, it uses a restrictive programming model, tailored to offer pass/fail responses for each test. Third, it only runs on locally constructed DOMs. The same restrictions make projects such as QUnit [17], Mocha [15], and YUI Test [12] unsuitable for large-scale experiments. In fact, QUnit, Mocha, and YUI Test do not offer even the limited parallelization that Jasmine provides.

Some tools, like Vows [10], offer parallelization, but are aimed only at testing JavaScript. These typically run on Node.js, which eliminates any DOM-interactive code from their domains, and naturally any URL-loaded pages.

Finally we consider web automation tools, program-controllable browsers like Ghost.py [9], PhantomJS [16], and Selenium [3]. None of these is explicitly a testing framework, so they do not offer convenient programming models for large-scale experiments. Also, Ghost.py and PhantomJS have no built-in parallelization. There is a variation on Selenium, Selenium Grid [20], that does offer parallelization. However, it is tailored for users who want to run the same small tests on multiple browsers and on multiple operating systems, usually to test a site's browser compatibility, rather than for users with many distinct results to collect as part of a large-scale experiment. Ghost.py, PhantomJS, Selenium, and other web automation tools like them, do all offer DOM interaction, the ability to run arbitrary JavaScript code, and the ability to load pages from URLs. However, in the case of headless WebKits, we found that their limited robustness reduced the ability to run arbitrary code.

All of the tools above lack a same-page guarantee. Ultimately most tools, being targeted towards developers, are intended for users who know their test pages will stay the same, or know how they will change. This makes them generally unsuitable for broader web experiments.

## 7. CONCLUSION

To this point, no JavaScript testing framework has been targeted at helping users test their web algorithms in controlled experiments. This paper has presented a testing framework for running large-scale, DOM-interacting controlled web experiments. It handles test parallelization, and permits users to run arbitrary DOM-interacting code on real webpages. Most importantly, it is the first framework to offer a same-page guarantee. This guarantee allows users to control for the fact that webpages change over time, in order to conduct fair experiments. Further, our framework's caching proxy server offers a better same-page guarantee than any existing proxy server. We believe that our framework represents an important step in web algorithm testing, bringing controlled web experiments within reach of a wider population. As the need for web-processing algorithms grows steadily greater, tools like this framework will help programmers handle the rapid and constant evolution of their webpage inputs, enabling them to build robust, empirically validated web algorithms.

## 8. ACKNOWLEDGEMENTS

This work is supported in part by NSF Grants CCF-1018729, CCF-1139138, CCF-1337415, and CCF-0916351, NSF Graduate Research Fellowship DGE-1106400, a grant from DOE FOA-0000619, a grant from DARPA FA8750-14-C-0011, and gifts from Mozilla, Nokia, Intel and Google. Additional thanks to John Kubiatowicz and Anthony Joseph.

## 9. REFERENCES

[1] Browser scripting, data extraction and web testing by iMacros. http://www.iopus.com/imacros/.
[2] sbarman/webscript. https://github.com/sbarman/webscript.
[3] Selenium-web browser automation. http://seleniumhq.org/.
[4] Alexa. Alexa top sites. http://www.alexa.com/topsites.
[5] Apache. Apache traffic server. http://trafficserver.apache.org/.
[6] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and customization of rendered web pages. UIST '05.
[7] Juliuz Chroboczek. Polipo: A caching web proxy. http://www.pps.univ-paris-diderot.fr/~jch/software/polipo/.
[8] Jasmine. Jasmine introduction-1.3.1.js. http://pivotal.github.io/jasmine/.
[9] Jeanphix. ghost.py. http://jeanphix.me/Ghost.py/.
[10] Vows JS. Vows - asynchronous BDD for node. http://vowsjs.org/.
[11] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. CHI '08.
[12] YUI Library. Test - YUI library. http://yuilibrary.com/yui/docs/test/.
[13] Moxie Marlinspike. New tricks for defeating SSL in practice. Presented at BlackHat, 2009.
[14] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: deterministic capture and replay for JavaScript applications. NSDI'10.
[15] Mocha. Mocha - the fun, simple, flexible JavaScript test framework. http://visionmedia.github.io/mocha/.
[16] PhantomJS. PhantomJS | PhantomJS. http://phantomjs.org/.
[17] QUinit. QUnit. http://qunitjs.com/.
[18] Selenium. Selenium IDE plugins. http://www.seleniumhq.org/projects/ide/.
[19] SquidCache. Squid: Optimising web delivery. http://www.squid-cache.org/.
[20] ThoughtWorks. Selenium grid. http://seleniumgrid.thoughtworks.com/.
[21] Duane Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE JSAC*, 16:345–357, 1998.