Automated Construction of JavaScript Benchmarks

Gregor Richards, Andreas Gal[†], Brendan Eich[†], Jan Vitek

S[3]Lab, Computer Science Dept., Purdue University
[†] Mozilla Foundation

Abstract

JavaScript is a highly dynamic language for web-based applications. Many innovative implementation techniques for improving its speed and responsiveness have been developed in recent years. Industry benchmarks such as WebKit SunSpider are often cited as a measure of the efficacy of these techniques. However, recent studies have shown that these benchmarks fail to accurately represent the dynamic nature of modern JavaScript applications, and thus may be poor predictors of real-world performance. Worse, they may lead to the development of optimizations which are unhelpful for real applications. Our goal in this work is to develop techniques to automate the creation of realistic and representative benchmarks from existing web applications. We propose a record-and-replay approach to capture JavaScript sessions which has sufficient fidelity to accurately recreate key characteristics of the original application, and at the same time is sufficiently flexible that a recording produced on one platform can be replayed on a different one. We describe JSBENCH, a flexible tool for workload capture and benchmark generation, and demonstrate its use in creating eight benchmarks based on popular sites. Using a variety of runtime metrics collected with instrumented versions of Firefox, Internet Explorer, and Safari, we show that workloads created by JSBENCH match the behavior of web applications.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Design studies; Measurement techniques; D.2.8 [*Metrics*]: Performance measures

General Terms Languages, Measurement, Performance

Keywords Reproduction, Repetition, Benchmarks

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

JavaScript's popularity has grown with the success of the web. Over time, the scripts embedded in web pages have become increasingly complex. Use of technologies such as AJAX has transformed static web pages into responsive applications hosted in a browser and delivered through the web. These applications require no installation, will run on any machine, and can provide access to any information stored in the cloud. JavaScript is the language of choice for writing web applications. Popular sites such as Amazon, GMail and Facebook exercise large amounts of JavaScript code. The complexity of these applications has spurred browser developers to increase performance in a number of dimensions, including JavaScript throughput [6].

Because browser performance can significantly affect a user's experience with a web application, there is commercial pressure for browser vendors to demonstrate performance improvements. As a result, browser performance results from a few well-known JavaScript benchmark suites are widely used in evaluating and marketing browser implementations. The two most commonly cited JavaScript benchmark suites are WebKit's SunSpider[1] and Google's suite associated with their V8 JavaScript engine[2]. The benchmarks in both of these suites, unlike real web applications, are small; V8 benchmarks range from approximately 600 to 5,000 lines of code, most SunSpider benchmarks are even smaller. Unrepresentative benchmarks may mislead language implementers by encouraging optimizations that are not important in practice and by missing opportunities for optimization that are present in the real web applications but not in the benchmarks. Weak benchmarks have had a negative impact on language implementations in the past. For example, the SPECjvm98 benchmark suite was widely used to evaluate Java [4] even though there was agreement in the community it was not representative of real applications. Dissatisfaction with SPECjvm98 led to the creation of the DaCapo benchmark suite, which includes realistic programs [1].

[1] `www2.webkit.org/perf/sunspider/sunspider.html`

[2] `v8.googlecode.com/svn/data/benchmarks/v6`

In previous work we measured the behavior of real-world JavaScript applications [13] and compared those results to similar measurements for the industry standard benchmarks. Figure 1 shows one visually striking example of the difference between the SunSpider benchmark suite and a real web application, in this case the Google search engine. The figures show the accesses performed on objects over their lifespan. Time is measured in events that have the object as a target for a read, add, delete and update operation. Object lifespans are normalized to construction time (shown as a vertical line in the graph). SunSpider is completely unrepresentative of Google's behavior. First, objects stay live for the duration of the benchmark. Second, the vast majority of operations are reads and adds. Third, once constructed, the "shape" of objects, i.e. their fields and methods, stay unchanged. Lastly, no fields or methods are deleted. None of these hold in the Google code. The benchmark completely misses out on the use of dynamic features of JavaScript such as object protocol changes and the extensive use of eval which we documented in [12]. Our claim in [13] was that industry benchmarks should not be used to draw conclusions about the performance of JavaScript engines on real-world web applications. Contemporary work came to the same conclusion [11]. At the time we did not provide an experimental validation of this claim. We do this in the present paper (see Figure 19).

The goal of this work is to provide developers with the means to create JavaScript benchmarks that are representative of the behavior of real-world web applications, so that these benchmarks may enable evaluating the performance of JavaScript implementations. In particular, these benchmarks should contain instruction mixes that retain the dynamism we have observed in web applications. There are multiple challenges that make the task difficult. Amongst them, many web applications are constructed dynamically by servers. Thus the same web page can have different behaviors at different times. JavaScript code can come in many forms: as embedded <script> tags in a HTML document, files that are downloaded by the browser, or dynamically constructed strings of text that are executed by eval. The code executed for any given web page is often browser-specific, with different features activated depending on the browser capabilities or special algorithms to circumvent performance issues in some JavaScript implementations. Moreover while JavaScript is a single-threaded language, JavaScript programs are exposed to a number of sources of non-determinism. Web applications are event-based applications in which mouse position and timing of user actions can affect the behavior of the program, timed events fire at different rates on different systems. All of these features render the task of creating repeatable performance benchmarks challenging.

In addition to the technical challenges, another feature of real-world JavaScript applications is their fluidity. The technology used to design web applications and the expectations

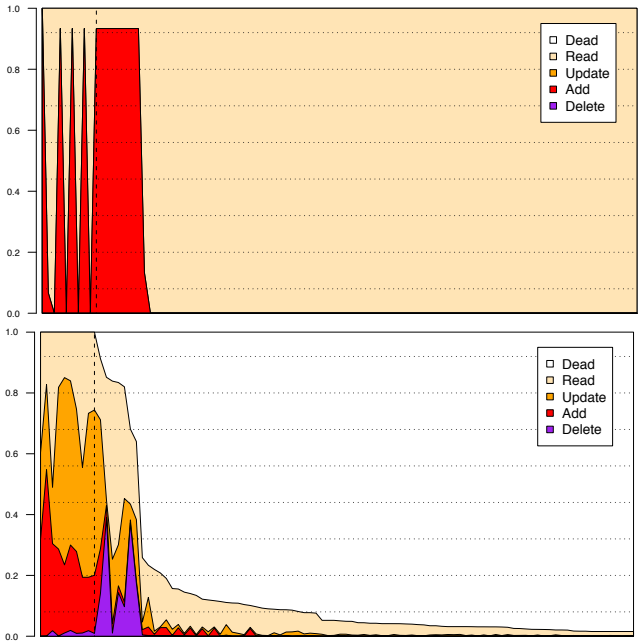


Figure 1. Object timelines. Comparing the operations performed on objects in industry standard benchmarks and web applications. Above, SunSpider. Below, Google.

of the users are evolving rapidly, towards richer and more behavior-intensive applications. This means that it is unclear whether one fixed benchmark suite can remain relevant for very long. Instead we argue that it is desirable to develop tools that will allow any web developer to create a set of benchmarks that capture the behaviors that are relevant to that particular developer at the time. Until now, no strategy for generating stand-alone, "push-button" replayable benchmarks has been proposed. Our tool, JSBENCH, fills this gap by automatically generating replayable "packaged" benchmarks from large JavaScript-based web applications. We achieve this goal by building a browser-neutral record-replay system. While such a system can be used for other purposes such as producing test cases or capturing crash-producing JavaScript executions in the field, in this paper we focus on using record-replay-based techniques to produce representative benchmarks for evaluating JavaScript implementations. Following industry practice, we focus on throughput-oriented performance measurements, leaving an assessment of responsiveness to future work. Furthermore, we designed JSBENCH to isolate the performance of JavaScript engines from other compute-intensive browser tasks such as layout, painting, and CSS processing. This allows for a head-to-head comparison of JavaScript implementations across different browsers. We aim to produce benchmarks that fulfill four requirements:

1. **Deterministic replay:** multiple runs of a benchmark should display the same behavior.

2. **Browser-independence:** a benchmark's behavior should not be affected by browser-specific features and should execute on all browsers.
3. **Fidelity:** benchmarks should correctly mimic the behavior of live interactions with a web application. As there is no *result* in a web page, we focus on the execution of events and changes to the web page.
4. **Accuracy:** benchmarks should be representative of the performance and non-functional characteristics of the original web applications.

Previous projects have either focused on recording client-side user behavior (e.g. [9]) or proposed browser-based instrumentation and recording techniques [11, 13]. Client-side recording can fail to intercept some of the JavaScript code and requiring a proxy to be present at the time of replay. Browser-based approaches do not help with the goal of synthesizing browser-independent benchmarks.

In summary, this paper makes the following contributions:

- We propose a browser-independent record system, called JSBENCH, for reliably capturing complex user-driven applications based solely on JavaScript source-level instrumentation and generating benchmarks to faithfully reproduce the applications' behavior.
- We describe how the record-replay approach can be used for producing deterministic replays of non-trivial programs and propose trace post-processing steps designed to achieve high fidelity.
- Through demonstration, we show that complex web applications can be successfully captured using JSBENCH with little effort.
- We demonstrate results for eight real, large JavaScript applications. Our evaluation includes a variety of runtime metrics pertaining to the JavaScript engine behavior as well as JavaScript-browser interactions, such as memory usage, GC time, event loop behavior, DOM layout and repaint events, etc. These were obtained with instrumented versions of Internet Explorer, WebKit and Firefox.

We emphasize that the benchmark generation strategy we enable is most suitable for comparing and tuning the performance of JavaScript interpreters and just-in-time compilers. The immediate impact of this work is to evaluate the impact of different implementation techniques for the language. Longer term, we expect these benchmarks, coupled with the data of [12, 13], to help language designers evolve the language. We explicitly did not aim to evaluate performance of web browsers. While our tool allows to capture some of that behavior, we leave the task of dealing with aspects such as responsiveness and rendering to future work.

JSBENCH is open source and available from:

`http://sss.cs.purdue.edu/projects/dynjs`

2. Related Work

At the core of JSBENCH, we have devised a technique for capturing a program, including its dynamically generated components, and replaying it in the absence of the surrounding execution environment (browser, file system, network connections, etc.) and to do so deterministically. The approach can be generalized to other languages and systems and has applications, including mock object generation for unit testing [3]. Replay techniques have been investigated in the past, mostly for debugging purposes. Cornelis *et al.* [2] and Dionne *et al.* [5] have surveyed and categorized replay-based debuggers. In Dionne's taxonomy, JSBENCH is a data-based automatic replay system as it records data exchanged between the program and its environment and requires no human-written changes to the source of the monitored program. Failure of a replay occurs if the program's interactions with its environment deviates from the recorded trace. While unlikely, this can happen due to implementation difference between browsers. A self-checking mode for each benchmark may be used to catch departures from pre-recorded traces. Related systems have been proposed. Mugshot [9] is a record-replay tool for JavaScript which aims for the highest possible fidelity, and as such recreates events exactly as they appeared in the original recording. Although suitable for debugging, this mechanism is too fragile for our goal of general-purpose replay, as it prevents the user from dispatching events and requires specialized functions to recreate every possible event that the browser could dispatch. The API's for dispatching browser events are inconsistent between browsers, and furthermore require careful tuning of the event infrastructure to assure that only the desired handlers are called. This creates browser-inconsistency and unpredictable overhead, both of which are unacceptable for benchmarking. Our system does not depend on recreating and dispatching true browser events, and as such relies only on the JavaScript language itself. This is simpler and less obtrusive. Another difference is that Mugshot aims to be an "always-on" system and thus must be extremely low-overhead and deal with privacy issues. JSBENCH can afford higher recording costs and needs not worry about confidentiality of user data. Mugshot also requires a proxy at the time of replay, which JSBENCH does not need. Ripley [14] replays JavaScript events in a server-side replica of a client program. DoDOM [10] captures user interaction sequences with web applications to allow fault-injection testing for detecting unreliable software components. While DoDOM captures sequences of external interactions with a web application for later replay, it assumes that later interactions will be with the original web site. Additionally, there have been several systems to instrument JavaScript at the source level. JSBENCH in particular is based on the same framework as AjaxScope [7] and Doloto [8]. However, our goal of having minimum impact on the behavior of the original code is quite different from these systems.

3. Record/Replay Principles and Requirements

A JavaScript program running in a browser executes in a single-threaded, event-driven fashion. The browser fires events in response to end-user interactions such as cursor movements and clicks, timer events, networks replies, and other pre-defined situations. Each event may trigger the execution of an *event handler*, which is a JavaScript function. When that function returns, the JavaScript event loop handles the next event. The timing and order of events is dependent on the particular browser, system, and other environmental factors. Thus, there is non-determinism due to the order in which events are processed by the browser. The same program will yield different results on different processors (the rate of processing events is different) and different browsers (the internal event schedule may be different). The JavaScript code interacts with the browser, the network and even indirectly the file system through a set of native operations that have browser-specific semantics. Figure 2 illustrates common sources of non-deterministic behavior in JavaScript-based web applications. Since every browser implements its own proprietary API's, and frequently these API's are undocumented, Figure 2 is necessarily incomplete, but from inspection and experience is a representative list of sources of non-determinism which are portable between browsers. Repeatability is further complicated by the fact that all web applications are interacting with one or more remote servers. These servers provide inputs to the program and code of the application.

Presentation	DOM objects event handlers
Network	XMLHttpRequest objects event handlers
File System	DOM objects (cookies)
Time	Date, setTimeout
User input	DOM objects event handlers
Nondeterministic functions	Math.random
Environment queries	document.location, navigator

Figure 2. Sources of non-determinism in JavaScript-based web applications. DOM objects mirror the layout of the web page, Listeners are used to associate callbacks to events, XHR objects provide asynchronous HTTP requests, setTimeout associates a callback to timer events.

In order to create benchmarks that are reproducible, these sources of non-determinism must be isolated. Thus, we are looking to produce benchmarks that somehow approximate, in a deterministic and browser-neutral fashion, these non-deterministic programs. Ideally we would want a deterministic program that faithfully and accurately reproduces the original program. Another challenge is that there is no clear notion of *output* of a web based application, no single result that it is intended to produce. To capture a web site and turn it into a replayable benchmark, we propose a record/replay approach with the following steps:

1. A client-side proxy instruments the original web site's code to emit a trace of JavaScript operations performed by the program.
2. The trace is filtered to get rid of unnecessary information;
3. A replayable JavaScript program, with all non-determinism replaced, is generated from the trace;
4. The program is recombined with HTML from the original web site.

While it is an attractive cross-browser approach, benchmark generation through JavaScript instrumentation has certain disadvantages. By introducing new JavaScript code into the replay (the code that removes non-determinism at the very least must be added to the original program), there is unavoidable perturbation of the original behavior. If we fail to fully instrument the code (such as would be the case when eval is called) on code that was not observed by the instrumentation proxy, certain parts of the program may not be recorded. We will now describe the properties and requirements for replayable programs.

Definition 1. *The execution state of a web application consists of the state of a JavaScript engine P and an environment E. A step of execution is captured by a* transition relation $P|E \xrightarrow{\alpha}_t P'|E'$ *where α is a label and t is a time stamp.*

As shown in Figure 3, the set of labels is split into labels representing actions initiated by the environment, α^E (either events or returns from calls to native browser functions), and actions performed by the JavaScript engine, α^P, which include function calls and returns, property reads and writes, object allocation, calls to native functions, etc. (These actions are modeled on TracingSafari [13], JSBENCH only captures the subset of events needed for creating replays.)

α^E	**Browser interactions**
EVT f, v	External event handled by function f
REP v	Return value v from an external call
α^P	**Trace events**
APP f, v	Call function f with arguments v
RET v	Return value v from a call
GET v, p	Read member p from object v
SET v, p, v'	Set member p from object v to v'
NEW f, v	Create an object with constructor function f
INV f, v	Invoke external operation f

Figure 3. Trace events. Operations performed by a JavaScript program and inputs from the environment.

Definition 2. *A trace T is a sequence $\alpha_1, t_1, \ldots, \alpha_n, t_n$ corresponding to an execution $P|E \xrightarrow{\alpha_1}_{t_1} \ldots \xrightarrow{\alpha_n}_{t_n} P'|E'$. We write $P|E \vdash T$ when execution of a configuration $P|E$ yields trace T.*

For any given program P, the same sequence of end-user actions (e.g. mouse clicks, key presses, etc.) can result in a different E due to timing and network latency issues. Different browsers will definitely lead to different environment behavior. Thus, requiring traces to be *exactly equal* is overly stringent. As there is no single output, we consider two traces to be equivalent if the display elements shown by the browser are identical. This notion of equality captures the end-user observable behavior of the program. We write $T|_p$ to denote the sub-trace of T matching predicate p. Thus, $T|_{\mathsf{DOM}}$ denotes a sub-trace of T composed only of SET actions on objects that belong to the DOM.[3]

Definition 3. *Two traces are DOM-equivalent, $T \cong_D T'$, if*

$$\mathsf{SET}\ \mathsf{v}, \mathsf{p}, \mathsf{v}, t \in T|_{\mathsf{DOM}} \implies \mathsf{SET}\ \mathsf{v}', \mathsf{p}, \mathsf{v}, t' \in T'|_{\mathsf{DOM}} \ \wedge$$

$$\mathsf{SET}\ \mathsf{v}', \mathsf{p}, \mathsf{v}, t' \in T'|_{\mathsf{DOM}} \implies \mathsf{SET}\ \mathsf{v}, \mathsf{p}, \mathsf{v}, t \in T|_{\mathsf{DOM}}$$

DOM-equivalence is not order preserving, because the order of event handlers executed by P may be different. A more complex equivalence relation could try to capture the ordering within event handlers and dependencies across event handlers, but a simple equivalence suffices for our purposes.

We are striving for determinism. This can be defined as yielding DOM-equivalent traces for *any* environment E. Intuitively, this is the case when the program is independent of external factors. Of course, deterministic programs running on *different* JavaScript engines may have slightly different traces due to browser optimizations, but they should be DOM-equivalent.

Definition 4. *A program P is DOM-deterministic if*

$$\forall E_1, E_2 : P|E_1 \vdash T_1 \wedge P|E_2 \vdash T_2 \implies T_1 \cong_D T_2$$

In order to create a replayable benchmark for some program P, the program must be run in an environment able to record its actions. We denote the recording $R(P)$ and write $\overline{T}^R$ for the trace obtained during recording. It worth to point out that $R(P)|E$ and $P|E$ will yield traces $\overline{T}^R$ and T which are not identical (but are DOM-equivalent). This is explained by the presence of additional JavaScript operations needed by the recording infrastructure appearing in the trace.

Definition 5. *Recording program P in environment E, written $R(P)|E \vdash \overline{T}^R$, results in replayable program P_R.*

The replayable program, P_R, is constructed so as to exhibit the following properties. First, the replayable program is fully deterministic, given the same environment it always yields the same trace, and secondly even in different browsers the traces are DOM-equivalent. We have not specified how to construct P_R, the details of this are an implementation choice.

Property 1. [Determinism] *$P_R|E$ always yields the same trace $\overline{T}$ and for any environment E', $P_R|E'$ yields a trace $\overline{T}'$ that is DOM-equivalent to $\overline{T}$.*

One technique that can be used to construct replayable programs is to avoid non-determinism by proxying. For any object v that performs a non-deterministic operations (e.g. a native call) in one of its methods, replace that object with a proxy, $\overline{\mathsf{v}}$, and memoize all of results returned by its methods during the original run, then use the memoized values in replay runs. Thus P_R could eschew non-determinism by always returning values obtained at record time for non deterministic calls. To abstract from the behavior of the browser, one can choose to record events issued by the browser and replay them in a deterministic and browser-agnostic order.

Of course, determinism is not a sufficient requirement as one could pick the empty program for P_R and get a really deterministic (and short) trace. The second required property of replays is that they preserve the behavior of the recorded trace.

Property 2. [Fidelity] *If $R(P)|E \vdash \overline{T}^R$, and $P_R|E \vdash \overline{T}$ then $\overline{T}^R \cong_D \overline{T}$.*

Fidelity is a property of recorded and replay traces stating the replay will issue the same DOM calls as the recorded trace. Fidelity is really a minimal requirement. It gives a sense that replay is "correct" with respect to the observed behavior of the recorded trace. But in order to get a benchmark that is a good predictor of the performance for original program we need to make sure that when shedding non-determinism, P_R has not been turned into a trivial sequence of memoized calls. What is needed is to make sure that replays are faithful to the computation performed in the original trace and not only it's DOM I/O behavior. Given a suitable definition of distance between traces, δ, the replay should be within the range of possible traces generated by the original program.

Property 3. [Accuracy] *If $P|E \vdash T$ and $P_R|E \vdash \overline{T}$, there is some environment E' such that $P|E' \vdash T'$ and the distance between the traces $\delta(T, \overline{T}) < \delta(T, T')$.*

Accuracy says that a trace generated from a replay is "close" to a trace that could have been generated from the original program under some environment. We deliberately leave the definition of distance open.

Another desirable property is to make sure that the computation performed by the replayable program P_R be comparable in running time to the original program execution. The execution time of trace T, of length n, written $\mathtt{t}(T)$, is $t_n - t_0$. Naively one would like that for a program

[3] This test can be implemented as v instanceof Node, but since the replay programs contain mock objects, we use (v instanceof Node || v.isJSBProxy()).

$P|E \vdash T$, and its replay $P_R|E \vdash \overline{T}$, the execution times of the traces be comparable $\mathsf{t}(T) \cong \mathsf{t}(\overline{T})$. This is unfortunately impossible to achieve while retaining determinism. Instead, $\mathsf{t}(T) > \mathsf{t}(\overline{T})$ is more likely. The replay trace suffers from *time compression* due to two main reasons. First, the original trace contains slack time between the last instruction of the previous event and the firing of the next event, $\mathsf{t}((\alpha, t), (\textbf{EVT}\ \mathsf{v}, t'))$. Second the compute time of native operations $\mathsf{t}((\textbf{INV}\ \mathsf{f}, \mathsf{v}, t), (\textbf{REP}\ \mathsf{v}, t'))$ can be substantial. The replay program does not preserve either. Slack time dependents inherently on the user and on the speed of processing the previous event. JavaScript does not have means to accurately control timing of event dispatches. Second, the execution time of native operations is highly dependent on the quality of the implementation of the browser. In Section 5, instead of simply comparing execution times, we argue that the replay has comparable dynamic attributes.

4. Creating Benchmarks with JSBench

To create a benchmark, users of JSBENCH need only point their browser to a web application hosted on some remote server. JSBENCH acts as a proxy between the user's browser and the server. Any JavaScript code loaded from the server will be intercepted by JSBENCH and rewritten in-place to an instrumented program with similar behavior. The instrumentation introduced by JSBENCH records a trace of the operations performed by the web application. Once the execution is complete, a post-processing pass is applied to turn that trace into a stand-alone JavaScript program consisting of the original HTML and a rewritten version of the source code in which dependencies to the origin server are removed and all non-deterministic operations are replaced by calls to mock objects returning memoized results. Figure 4 summarizes this architecture and the rest of this section discusses it in more detail.

4.1 Recording JavaScript executions

Before a piece of JavaScript code is handed to the browser for execution, JSBENCH instruments it, rewriting all function entries and exits as well as field access and related operations, as detailed below.[4] The purpose of the instrumentation is to create, as a side effect of executing the application, a trace of the executed operations. There are many potential mechanisms for observing the behavior of JavaScript programs. We have based JSBENCH on source-level instrumentation for two reasons:

- Source-level instrumentation is *portable* across different browsers and different versions of the same browser, users will thus be able to create benchmarks on any browser.

[4] Code in this section is simplified for readability and to spare the reader the gruesome details of assuring that JSBENCH's variables are never shadowed, and other nuances of JavaScript that are not directly relevant.

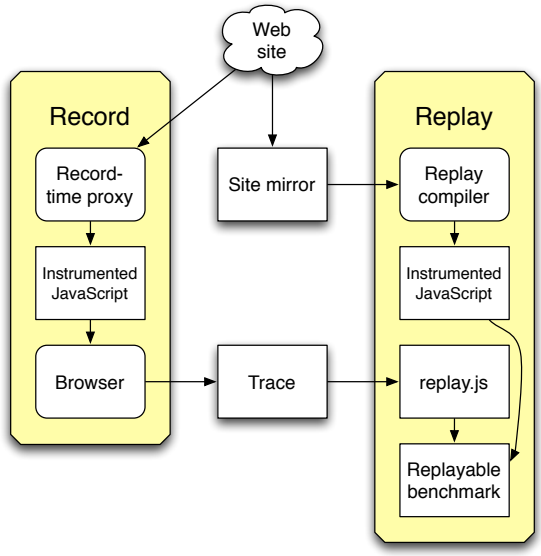


Figure 4. System architecture. JSBENCH acts as a proxy between the browser and the server, rewriting JavaScript code on the fly to add instrumentation and finally creating an executable stand-alone application.

DEC o	Declare object or function o exists.
GET o, f, v	Field o.f has value v
SET o, f, v	Field o.f is updated to v
APP o, [o1 ...], v	Calling function o with return value v.
NEW o, [o1 ...], o'	Creating object o' with constructor o.
EVT g, [o1 ...]	Function g fired as an event with arguments [o1 ...] . g may either be a static function, or the result of a CTX event.
CTX o, g, [f1 ...], [v1 ...]	The function o refers to the original static function g with variables [f1 ...] in its closed context referring to values [v1 ...] .

Figure 5. Record-time trace statements.

- The JavaScript language *evolves at a slower rate* than its implementations.

JSBENCH uses proxy-based instrumentation of JavaScript code to record an execution trace that contains all the information needed to create an executable replay of that trace [7]. The trace generated by JSBENCH consists of a sequence of labeled trace statements, as detailed in Figure 5. DEC declares that a pre-existing JavaScript object has been encountered, GET and SET indicate reads and writes of properties, APP indicates a function call, NEW marks the construction of a new object, EVT denotes the firing of an event, and CTX denotes the construction of a closure. Not all operations need to be logged. Indeed, only operations that introduce non-determinism must be recorded. For those, JS-

BENCH records enough information to reproduce their behavior deterministically. JSBENCH will record calls with their arguments as well as their results, and replace any object that has non-deterministic operations with a *memoization wrapper*. A memoization wrapper is a small JavaScript object which contains a reference to the original object and a unique identifier. The purpose of the memoization wrapper is to record all operations performed over the object and any objects or functions it refers to. We avoid creation of multiple memoization wrappers for the same original object leveraging the dynamism of JavaScript. JSBENCH extends wrapped objects by adding one extra field that holds a back pointer to the wrapper. Some native objects can not be extended. In this case, a reverse reference is not stored, a warning is produced, and each access to the object will create a new wrapper. Every time a memoization wrapper is created, a DEC event is logged in the trace. The behavior of certain functions must be memoized as well. A *memoizing function* is similar in principle to a memoization wrapper. It stands between the code and the real function, and its behavior when called is identical to that of the function it memoizes, but it also records an APP event. The reproduction of memoized functions at replay time must be capable of producing the same argument-to-value mapping as was seen by APP events. Constructors are just a special case of functions. When a memoizing function is called with **new** (and thus, as a constructor), a NEW event is generated, but otherwise the behavior is the same. JSBENCH makes sure that before any JavaScript code is allowed to run, a JSBENCH-generated function is run which replaces all of the objects and functions that we have identified as introducing non-determinism with memoization objects and memoizing functions. This function also produces DEC and GET trace events for each replaced object.

4.1.1 Instrumenting field accesses

JSBENCH needs to instrument field accesses. This is for two reasons: reads of DOM and other wrapped objects will be memoized for replay, and writes to these objects may be checked at replay time to ensure fidelity. Reads are denoted by GET events in the trace, writes by SET events. JavaScript's highly-dynamic nature makes record-time memoization a straightforward task: all object member accesses are rewritten to use a logging function, which returns an object containing the appropriate member. The JavaScript expression x[exp], which uses associative syntax to access the field named by the value of the expression exp of object x, is rewritten into:

```
( memoRead( x, t = (exp) ) )[ t ]
```

where t is a unique temporary variable, used to assure that side effects of exp are not repeated. The more conventional syntax for reading fields, x.y, is syntactic sugar for x["y"], and is rewritten appropriately.

The memoRead() has a somewhat baroque definition as a direct consequence of the semantics of JavaScript. In JavaScript, functions take an extra argument to a context object. Within the body of the function this argument is the implicit **this** reference. A call of the form x.f() will execute function f with **this** bound to x. A call of the form x[f]() must also bind **this** to x in f. But, surprisingly, the following z=x[f];z() will not. Our translation preserves this semantics so, memoRead does not return the value of the property rather it returns an object that has the desired property. Consider the following access to x["f"]:

```
x["f"]   ⇒
memoRead(x,t="f")[t]   ⇒
{f:wrap(x["f"])}["f"]
```

This design also allows unwrapped objects to be returned directly, while still preserving call semantics:

```
x["f"]  ⇒  memoRead(x, t="f")[t]   ⇒  x["f"]
```

The translation replaces the original expression with a call to memoRead() and, assuming x is wrapped, memoRead() returns a newly created object with a property f referring to a wrapped value. More precisely the memoRead() function behaves as follows. It checks if x is a memoization wrapper object. If not, it simply returns x. If it is, then the wrapper has a reference to the original object, the operation is forwarded to that object and a GET event is logged. The return value depends on the type of the value referenced by the requested property:

- A primitive: The wrapped object is returned without further processing.
- An object: if the object has not previously been wrapped in a memoization wrapper, a memoization wrapper is created for it and the associated DEC event is logged. A new object containing a mapping of the expected property name to the wrapper is created and returned.
- A function: a memoizing function is created which wraps the original function. A new object containing a mapping of the expected property name to the memoizing function is created and returned. The memoizing function is capable of determining when it is called with the returned object as the context (value of **this**); in this case, the original function is called with **this** as x. Otherwise, the original **this** is passed through.

Assignments are implemented similarly, with a memoWrite() function which logs a SET event for memoization wrapper objects, and performs the associated write. It can also deal with operators such as += by an optional argument which describes which operation is used to update the argument. This case produces both a GET event of the original read, and a SET event of the new value.

Other related operations worthy of note include operator in, statement for(...in...), equality-related operators and instanceof. They are replaced by memoization-wrapper-

```
1 function onloadHandler() {
2   myvalue = document
3     .getElementById("input")
4     .value;
5 }
```

(a) Source code

```
1 EVT onloadHandler, [window]
2 DEC o1
3 GET window, document, o1
4 DEC f1
5 GET o1, getElementById, f1
6 DEC o2
7 APP f1, [o1, "input"], o2
8 GET o2, value, "Hello!"
```

(b) Recorded Trace

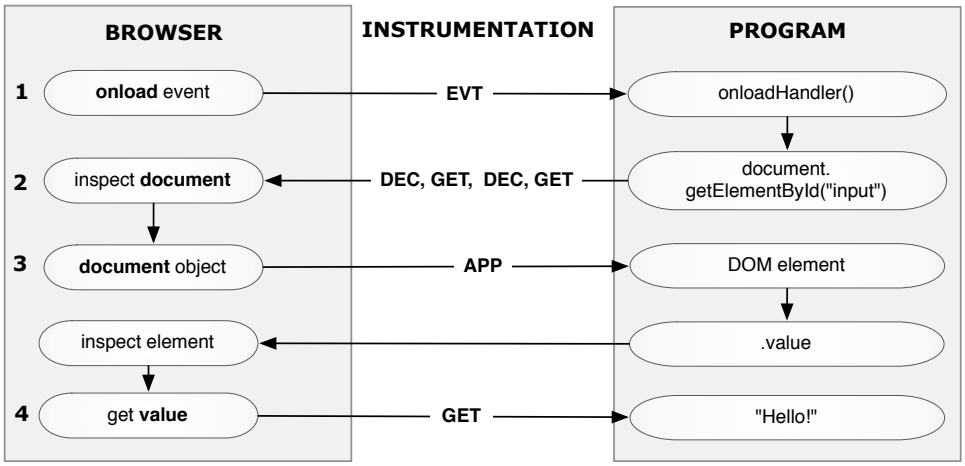


Figure 6. Example. Source code to the trace and browser and JavaScript interactions.

aware versions. This is a much simpler task, as none of these return objects, so no new wrappers need be generated. y in x is replaced by memoIn(y, x), and for (y in x) is replaced by for (y in unwrap(x)). Both of these functions simply perform the action of the operation they replace on the wrapped object. Equality operators are replaced by unwrapping versions, such as unwrap(x)==unwrap(y).

4.1.2 Instrumenting functions

JSBENCH instruments every function entry and exit point. In the browser model, a program execution consists of sequence of single-threaded event handler calls. As any function can be used to handle events, we need to identify which functions are being invoked as event handlers and memoize the arguments of these functions for replay. JSBENCH modifies every function in the source code to check if its caller is another JavaScript function. If this is not the case, then JSBENCH deduces that the function has been invoked by the browser in reponse to an event happening. Every function in the source is instrumented as shown in Figure 7. The logEvent() adds an EVT event corresponding to the current function invocation to the trace. In addition to memoizing the event handler arguments, if the event handler is a closure, then JSBENCH must also memoize the environment of that closure. This is necessary because at replay time the closure may try to access variables in its environment. As it impossible to name the particular closure called as a handler, the replay instrumentation wraps the function such that it can recreate part of its context. The logContext() function generates a CTX event, as well as a unique closure ID for this particular instance of this function. The closure context alluded to in the above code is an object literal which maps the names of all variables which are closed over (determinable statically since JavaScript is lexically scoped) to their current values. For instance, if f closed over variables x and y, the closure context object literal would be {x:x, y:y}. This strategy works can lead to over-memoization. The post-processing used to alleviate this is discussed next.

```
function f (...) {
  if (!callerJS) { // event handler
    callerJS = true; var ret;
    try {
      var hid = logContext(f,{/* closure context */});
      var wthis = memoizationWrap(this);
      var args = arguments.map(memoizationWrap);
      logEvent(hid, wthis, args);
      ret = f.apply(wthis, args);
    } catch (ex) { callerJS = false; throw ex; }
    callerJS = false; return ret;
  } else { /* original function body*/ }
}
```

Figure 7. Example of function instrumentation.

4.1.3 Example

To illustrate in the inner workings of JSBENCH, we provide an extended example. Figure 6 shows the browser operations on the left hand side and JavaScript engine operations on the right hand side. To describe the interaction, we start with a piece of JavaScript source code in Figure 6(a). Figure 6(b) shows the trace that JSBENCH logs as part of recording the execution of this code.

1. **onload:** onLoadHandle is set up as a listener such that it will be called by the browser when the load event occurs. The first step of the recording process signified by the EVT label records the presence of an external event that invokes the onLoadHandler handler.

2. **Calling getElementById:** The next step involves two references to objects not defined in the program. The trace system creates temporaries for them: o1 (document object provided by the browser) and o2 (the DOM element corresponding to input). Additionally, the native function document.getElementById is referred to as f1. Two GET labels in the trace reflect the relevant state of the program heap.
3. **Returning the DOM element:** Next, we create a trace object o2 for the result of getElementById call and record a APP label identifying this fact.
4. **value property lookup:** The last step records the text string which corresponds to property value of o2. This is recorded with the last GET label.

Once the trace has been generated, converting to the replay script shown in Figure 8 is a syntax-directed translation process. Lines 2 and 6 initialize the two objects used for replay. Line 4 does the same for f1. Bodies of all such functions use memoization tables to map inputs to outputs, as illustrated by the cases setup in line 8. GETs in the trace result in heap assignments in lines 3, 6, and 9. Finally, each event in the trace corresponds to a function call such as shown in line 10.

```
function replay() {
  var o1 = {};  window.document = o1;
  var f1 = function() {
             return lookupCase(f1, this, arguments); }
  o1.getElementById = f1;
  var o2 = {};  f1.cases[2][o1]["input"] = o2;
  o2.value = "Hello!";
  onloadHandler.call(window);
}
```

Figure 8. Example replay function.

4.2 Replaying recorded traces

The instrumentation of existing code for replay is far lighter than record instrumentation. There are only two instrumentation steps. In some cases it is not possible to override variables in JavaScript, e.g. certain browsers do not allow document to be overwritten, and in these cases references to such variables must be renamed in the original code. This is accomplished simply by replacing all instances of the identifier with an automatically-generated unique identifier. Not all static functions have a global name. But for the replay to be able to call them, there must be a reference to them which the replay code can access. Those functions which are referred to by EVT events are instrumented to store a reference to the function in an object owned by JSBENCH. For instance, if a function func is defined, and is used in an EVT event with ID f1, then a statement such as window.handlers.f1 =func is added to the program to retain the reference.

4.2.1 Generating the replay code

At replay time, the lightly-instrumented original code is combined in file called replay.js, which is generated from the trace. All of the objects which were memoized at record time will be replaced with mock objects at replay time which expose the same API as was used in the original source. Memoized objects are replaced with mock objects which have all of the recorded members (all members seen by GET events) with their original values, and memoizing functions are replaced by mock functions, which map all of the arguments seen in APP events to the return values seen. To generate replay.js, all events are grouped with their preceding EVT event, the event handler that caused the relevant action. Events which are grouped with an EVT generate code which precedes the code generated for EVT itself, to ensure that the created mock objects expose the API which the handler uses. Figure 9 shows the JavaScript code generated for each trace statement. DEC, GET, APP and NEW events are used to construct mock objects and functions. DEC events are replaced by creating an object named by their unique ID. Non-redundant GET events are replaced by an assignment to the relevant mock object. Each mock function generated contains an associated mapping between arguments and return values. APP and NEW events are replaced by adding new entries to this mapping. SET events do not need to be replayed since they represent a behavior performed by the replayed JavaScript code, and not a source of non-deterministic behavior, but may optionally be replayed as assertions to verify that the field sets actually occur in replay.

DEC o (for objects)	var o = {};
DEC o (for functions)	var o = **function**() { **return** callCase(o,**this**,arguments);}
GET o, f, v	o.f = v;
APP o, [o1...on], v	o.cases[n][o1]...[on] = v;
NEW o, [o1...on], o'	o.newCases[n][o1]...[on] = o';
EVT g, [o1 ... on]	g. call (o1, ..., on);
CTX o, g, [f1 ... fn], [v1 ... vn]	o = g(v1, ..., vn);

Figure 9. Replay statements.

4.2.2 Trace Post-processing

Our tracing framework captures a large amount of information, and much of it is easily determined to be unnecessary during replay. Although a trivial conversion of the trace into a replay function is feasible and can produce semantically correct results, the overhead of such naïve conversion is too high. To preserve accuracy of the captured trace, the trace is processed and filtered in various ways before producing the final replay code. The most important post-processing

task is trimming. The record-time instrumentation memoizes some objects which ultimately do not need to be generated as mock objects in the replay. In particular, since we memoize all variables in the context of each closure used as an event handler, recreating all the relevant mock objects would be both expensive and unnecessary. To restore true interaction with a given object in a trace, it is possible to follow the variable and its dependencies through the trace, determine all objects which are ultimately attained by references from the variable to be removed, and remove them. All EVT events, when logged during recording, have an associated CTX event describing the event's closure context. The replay implementation wraps functions which are referred to by at least one CTX event such that their closure context may be reproduced. For instance, a function f with ID f1 which closes over x and y but modifies neither is wrapped as follows.

```
var f = (window.handlers.f1 = function(x, y) {
    return function() { /* original body */ }
})(x, y);
```

However, quite frequently the closure context never changes, making this wrapping an unnecessary expense. As such, all CTX events which correspond to the same static function are compared, and those variables which never change are removed. Any resulting CTX events which refer to no variables at all are removed, and associated EVT events are then free to refer to the static functions directly. Such EVT events do not generate wrapped functions to regenerate closure context, and as such have much lower overhead. In our experience, the vast majority of CTX events are removed from the trace.

4.3 Practical Challenges

While the principles of memoization are outlined above, many other issues had to be addressed before JSBENCH was able to record and replay large real sites.

Missed and Uninstrumented Code. There are two primary means by which uninstrumented code can be run: The proxy may have failed to recognize and instrument it, or it may have been generated entirely by JavaScript. The solution to the former case is simply to fix the instrumentation engine in the proxy, but the latter case is more difficult to contend with; JavaScript's semantics make it impossible to override the eval function, so there is in general no way to reliably ensure that generated code is instrumented. We have no solution to this problem, but have observed that it does not substantially reduce the correctness of the technique because:

- Most complicated eval code is loaded through XHR requests, and so will be instrumented.
- The remaining eval code tends not to access memoized objects.

Neither of these properties are intrinsic to the language however, and so it is possible for real benchmarks to function improperly due to uninstrumented code.

Failure Pruning. Our system is imperfect, and on some sites it fails to create a workable replay. Although ideally this would be solved by fixing whatever bug led to the replay failure, it is also possible to identify which events fail (by catching exceptions in the generated replay code) and pruning them from the log, thereby generating a working replay that captures less of the original behavior.

Closures. Although JavaScript's event dispatch semantics generally passes only simple and trivially-memoizable objects as arguments to event handler functions, the functions themselves can be closures, and as such may have implicit dependencies on arbitrarily-complex data in their closure contexts. Furthermore, it is impossible purely within JavaScript code to get a reference to a particular instance of a closure without having bound it to a name. Binding every closure to a name is impractical due to the high runtime cost of the mechanism. Furthermore, such a solution would be fragile to any inconsistencies between record and replay. Instead, we use our memoization techniques to capture all closed variables, but regenerate only those that change at runtime.

Mirroring. Making redistributable and replayable recordings requires mirroring the web page, as otherwise running the benchmarks would necessitates setting up the browser's HTTP proxy, an unacceptable requirement for performance testing. Mirroring dynamic web pages is a research problem in and of itself, but is not within the scope of this paper. We used three techniques to mirror web sites, each of which worked on particular sites:

- **Static mirroring.** Download tools such as wget have the ability to mirror web pages, but as they are not browsers and therefore cannot mirror dynamic content.
- **Dynamic mirroring.** Browsers such as Firefox and Chrome have the ability to save "whole" web pages by saving everything from the browser's internal state, rather than the original files. This has the disadvantage that dynamic changes made to the web page by JavaScript code will be saved, which can conflict with the replay code which will redo their effect.
- **Cache mirroring.** The proxy used for instrumentation is also capable of caching. The cached files can be extracted from the proxy for mirroring purposes. This has the advantage of including every referenced file, but the disadvantage of requiring manual remapping of the files to their new locations on disk.

Harness. To simplify running benchmarks generated by JSBENCH, the mirrored site is combined with replay.js and is placed in a separate iframe. The replay function is then called and the execution time is timed using the JavaScript

Date functions. Our harness uses a push-button approach to run all the recorded sites, similar to the setup of SunSpider benchmarks.

4.4 Replay Modes

While we are focusing on JavaScript throughput, one of the strengths of the JSBENCH approach is that the level of non-determinism in the replay code can be dialed up to evaluate other browser features. We have implemented three dimensions of customizability.

Event loop generation. In JavaScript, execution unfolds by calling a series of event handlers in a single thread. When replaying the event handlers, the challenge is to invoke them in a manner similar to the original in-browser execution. JSBENCH supports several options:

- All handlers may be collapsed into one, resulting essentially in one very long event. This approach does not exercise the browser's event loop, deviating from the original execution.
- The original events may be recreated by createEvent, then fired by dispatchEvent. However, this mechanism does not integrate with our memoization system, and has poor inter-browser compatibility. These methods are uncommon in real code, so being sensitive to their timing may produce unrealistic benchmarks.
- Event handlers may be invoked using the setTimeout function, passing the handler closure as the callback argument and 0 as the delay. Ideally this would simply yield to the browser and fire as soon as possible, but in fact most browsers cannot time out for less than a particular OS- and browser-dependent amount of time, so most of the execution time would be spent idle.
- The postMessage mechanism, although intended for inter-domain and inter-frame communication, also yields to the browser's event loop to fire the message handler. On most browsers, there is no idle time between placing a message with postMessage and the handler firing, but other browser jobs can be performed. postMessage, however, is extremely uncommon in real code, and on some browsers the implementation is slow. In spite of these limitations, it is currently the most reliable mechanism available which yields to the browser event loop.

DOM Removal. If we wish to create a JavaScript benchmark that can be run outside the context of a browser or that exercise only the JavaScript engine, we need to remove accesses to DOM objects. As such, our record-time instrumentation memoizes DOM objects. Objects generally considered to be part of the DOM are those accessible by traversing members of the document object. The replays created will normally have no interaction with the DOM, as mock objects will be put in its place. To restore DOM interaction, we need only to trim document and its dependencies from the trace.

Mock Object Collection. Because every access to every memoized object is traced, the exact lifespan that each of these objects needs is known. As such, a simple processing procedure guarantees that mock objects are created as late as possible, and all references to them held by the replay code are removed as early as possible. This optimization trades time for space: without it, all mock objects would be alive for the entire duration of the replay, taking a lot of unnecessary space, but the process of creating and destroying them would not be part of the replay proper, removing that time from the recorded time of the benchmark.

5. Empirical Evaluation

This section provides a detailed empirical evaluation of JSBENCH as well as evidence supporting our claims about the quality of the benchmarks created with our tool. The empirical evaluation relies on two different research infrastructures which we extended for this paper.

1. TracingSafari is an instrumented version of the Safari 5 browser (WebKit Rev. 49500) based on [13] which is able to record low-level, compact JavaScript execution traces. We use it to compare the behavior of replays to live traces. Even though TracingSafari only runs in interpreter mode, the instrumentation is lightweight enough to be barely noticeable on most sites.
2. FirefoxMem is a heavily instrumented build of the Firefox 4.0b2 browser that produces exhaustive information about memory and object lifetimes. The cost of memory tracing is rather prohibitive, but FirefoxMem provides a very detailed picture of memory usage.

We used an unmodified version of Internet Explorer with ETW (low-overhead event tracing) enabled. We also used several different versions of popular browsers, listed in Figure 10, in order to highlight certain properties of the benchmarks generated by JSBENCH. Unless indicated otherwise, to obtain the experimental results presented in this section were obtained on an HP xw4300 Workstation (Intel Pentium 4 3.6 GHz machine with 2 Gigabytes of memory) running Windows 7 Enterprise 64-bit operating system.

Short name	Browser	Browser version
IE8	MS Internet Explorer	8 (8.0.7600.16385)
IE9	MS Internet Explorer	9 Preview 4 (9.0.7916.6000)
FF3	Mozilla Firefox	3.6.8
FF4	Mozilla Firefox	4.0b2
Chrome5	Google Chrome	5.0.375.99
Chrome6	Google Chromium	6.0.492.0 (Rev. 55729)
Opera10	Opera	10.61.3484
Safari5	Apple Safari	5.0.1 (build 7533.17.8)

Figure 10. Browsers used for measurements.

5.1 Web Applications

JSBENCH is capable of creating replayable workloads from most websites, but not all of these will be useful in practice. We have identified four properties for good candidates for benchmarks. *Reasonable execution time:* A short-running benchmark is subject to more perturbation due to system conditions than a long-running benchmark. Ideally the interaction with the web page should be long enough that the generated benchmark produces consistent results. *Meaningful interaction:* One can easily produce a long-running benchmark on many sites by simply waiting several minutes while timer events fire, then measuring those timer events. However, this is not a realistic interaction with the site. To be comparable to the real site's expected behavior, the interaction should be representative of a typical end-user's experience. *Limited overhead:* The amount of overhead introduced by JSBENCH depends on the number of proxies that have to be introduced. On a site for which the replay introduces many mock objects, the time taken to create skew the results. Many browsers come with profilers that can be used to estimate this overhead; the benchmarks we have presented all had an overhead of under 10%. *Generality:* The site chosen for recording should have behavior which is representative, so that its results generalize a significant segment of web applications.

We have constructed eight benchmarks for this paper, listed in Figure 11. The first two are small programs that are self-contained JavaScript applications. `sibeli.us` is an arcade-style game and JSMIPS is a MIPS emulator.[5] Both are small single-file JavaScript programs. The latter six represent a large class of widely-used web applications, according to alexa.com. We do not claim that this is a definitive set of benchmarks, selecting such a set will require further experimentation, community involvement and, in order to distribute the suite, consent from web site's owner. The power of JSBENCH comes from the fact that anyone can take their own benchmark and package it.

Benchmark	Bytes	Files	LOC
sibeli.us	19K	1	650
JSMIPS	108K	1	4,036
amazon.com	1,065K	10	23,912
microsoft.com	791K	21	17,241
bing.com	39K	4	1,872
maps.google.com	572K	8	5,318
economist.com	239K	5	7,129
msnbc.msn.com	621K	11	4,273
total	3,4564K	61	60,585

Figure 11. Summary. Information about original sites.

[5] `http://codu.org/jsmips`

For web applications we have captured user interactions of length around 1 minute. The sites that were selected, amazon, microsoft, bing google, economist, msnbc, are all among the top 100 US web site and according to [13] exhibit representative dynamic behavior. The sizes and complexity of these sites varies between 1,8 KLOC and 23 KLOC, as measured in lines of JavaScript code. As a point of comparison, the entire SunSpider and V8 benchmark suites have 13,963 and 11,208 LOC, respectively, being composed of small, well-behaved applications [11, 13].

5.2 Determinism

We have evaluated *determinism* of our benchmarks both within the same browser and across browsers and found no difference in the behavior of each benchmark. They run in a completely deterministic fashion and are DOM-equivalent.

Determinism of the replays across browsers hinges on hiding browser-specific behavior from the replay code by the means of mock objects. We experimented with the impact of removing the memoization around the navigator object which is used by JavaScript code to detect the browser. This will result in each browser behaving slightly differently if the application contains browser "sniffing" code — code specialized to a particular browser or version. Figure 12 shows the result of a replay instrumented to output the call path of the program. In this run, there are 10,773 function calls. Column 2 shows that if we memoize the navigator object, the replay is fully deterministic, as demonstrated by an identical call trace. Column 3 of the table shows the number of *differences* in the call trace obtained when we record with Firefox and replay with a different browser. Note that the difference numbers are all quite small, representing only a tiny percentage of the overall trace.

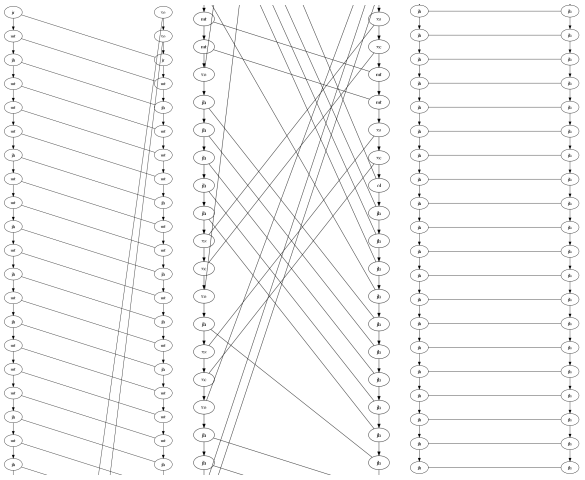
	Replay mode	
Browser	Real	**Poison**
IE 8	0	11
Firefox 3	0	0
Chrome 5	0	2

Figure 12. Browser differences. (economist.com)

5.3 Fidelity

One measure of fidelity is to verify that any DOM call observed during trace recording is present, with identical arguments, at replay time. This is true by construction of our trace and we have validated it experimentally.

Another measure of fidelity is to line up events fired in the recording and in the replay. JavaScript programs usually have several load-time events firing, many XHR events firing, then a sequence of user interaction events firing. We compare the event firing behavior. To do so, we collect event firing traces and post-processed them to correlate the events and their order of dispatch. We observed perfect fidelity, with all events appearing in record and replay traces, but some difference in ordering due to timing issues. Since the traces



(a) Load-time (b) Time-based events (c) Quiescence

Figure 13. Matching events. Comparing real execution and replay (amazon; IE9).

are about 2,000 events each, we cannot show them fully. Instead, we compare three representative segments of the trace separately. The result of this experiment is shown in Figure 13. Each oval represents an individual event, with real events on the left and replay events on the right.

(a) Load-time. The initial portion of the two traces match up quite well, except for the fact that they are offset by two "stray" XHR events happening in the replay that happen later in the real trace. This is an example of browser scheduling non-determinism.

(b) Time-based events. This segment is taken from the middle of the trace, when various timers that run as part of standard Amazon.com execution kick in. Unsurprisingly, with timer-based, XHR and onload events being fired by the browser's scheduler, the real and replay events can be scheduled in a very different order, as shown in the figure.

(c) Quiescence. This segment corresponds to the end of the trace and a state of quiescence for this site. The traces match up perfectly.

5.4 Accuracy

Comparing the behavior of the replay with the original program is a bit more tricky. A replay P_R has been obtained by running an instrumented program, thus it is conceivable that the behavior observed at recording, $R(P)$ is significantly different from an un-instrumented run of the original program P. While, ideally one could compare traces, $\delta(T, \overline{T})$, our infrastructure can not give us a trace of the original program without substantially perturbing the very characteristics we want to observe. So instead of measuring the distance between traces, we will argue for accuracy by observing a number of properties of original and replay executions and argue that they have sufficient similarities so that replays can be used as predictors of performance of the original application.

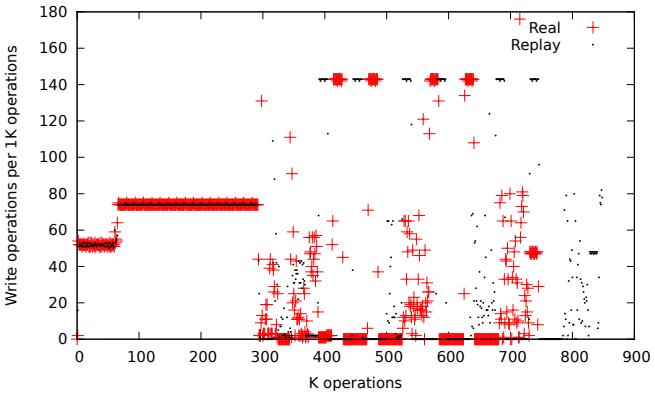


Figure 14. Write accuracy. Each point on x-axis represents one thousand bytecodes executed by the JavaScript engine. The y-axis gives the absolute number of object property writes performed in each 1K window. The maximum deviation observed over multiple run was 10.4%. (msnbc; TracingSafari).

As a first approximation of replay accuracy, we provide a high-level view of the updates performed by the benchmark on non-DOM objects. While fidelity ensures that all DOM updates performed in the recording will also happen at replay, it makes no guarantees about other writes. Figure 14 plots the number of writes that are performed in in a window of one thousand bytecodes. We compare an original (non-instrumented) run of msnbc with a run of the replay program. The data is obtained using TracingSafari as it has a non-intrusive (browser-specific) recording mechanism. Visually, it is clear that original and the replay line up, but are not identical. This is expected as any non-instrumented run will have different numbers of timer events, different order of events, and the replay has mock objects. We measure the difference of between the original and the replay trace using normalized root-mean-square deviation (NRMSD).[6] For five real and five replay runs, the maximum NRMSD is 10.4% which suggests that the replay are generally close to original runs in terms of the update operations they perform. The NRMSD between replay runs is always 0% (attesting to their determinism).

To get another reading on replay accuracy, we measured the internal operations performed the JavaScript engine during execution of a replay and compared it with an original run. For this measurement we used the ETW, a low-overhead tracing framework supported by Internet Explorer. ETW let us measure the number of invocations of the JavaScript parser, the bytecode compiler, the native code generator, other calls to the engine, and calls to the DOM. Fig-

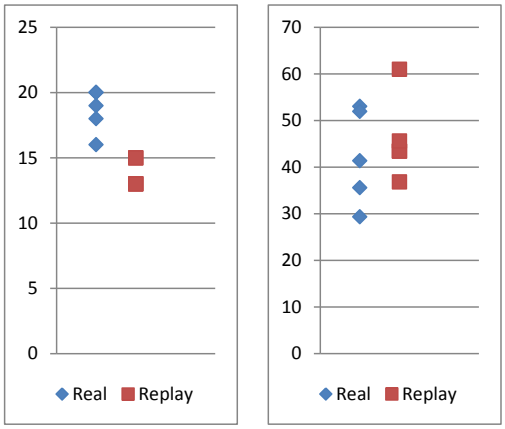
[6] NRMSD is a common statistical measure of the deviation between functions; however, it is not ideal as it has no ability to contend with repeated or re-ordered events.

Behavior	Real		Replay		Mean
	Mean	Std. dev.	Mean	Std. dev.	diff.
Parsing	20	0	20	0	0
Bytecode Gen	20	0	20	0	0
Native Code Gen	32.4	2.6	32.4	3.8	0
Calls to JS engine	2,540.6	36.86	2,525	0	15.6
Calls to DOM	62,032	566.35	61,753	0	279

Figure 15. Service tasks. Calls to the JavaScript engine over five runs of the real application and its replay. (amazon; IE9).

ure 15 summarizes the results of five different run of the original amazon site and five runs of the replay. As can be readily observed, different real runs have make slightly different number of service requests on the JavaScript engine. The replay on the other hand is deterministic. The last column gives the absolute difference in the means. This difference is rather small and within the standard deviation. The number of times parsing and bytecode generation is invoked is exactly identical in both original and replay.

ETW also allows us to measure the time spent garbage collecting. It is important to make sure that our replay mechanism does not fundamentally affect the memory behavior of the program. For this we report in Figure 16 the number of calls to the garbage collector in each run of the real and replay program and the time spent in GC. The number of GC cycles is slightly smaller in the replay runs but the amount of time actually spent in GC ranges from similar to slightly higher, which is easily explained by the introduction of mock objects.



(a) GC Cycles (b) GC Time (ms)

Figure 16. Garbage Collection. Comparing GC calls and execution time between real and replay. (amazon; IE9)

To get a better understanding of the memory usage and impact of mock objects we used FirefoxMem to record the JavaScript heap growth over time. Figure 17 shows the size of heap over time. It compare five real runs to five replay runs. The overall behavior is similar although one can observe time compression as the replays complete faster than the real runs. The replay runs with heap size 6.8% smaller than the original program, a reduction due to the fact that mock objects are smaller than the objects they are replacing. Though, this need not be the case.

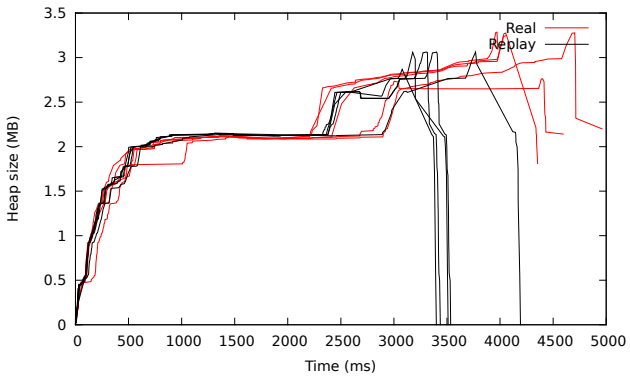


Figure 17. Memory usage. The y-axis gives the size of the JavaScript heap. (economist; FirefoxMem)

We observed that replay experience *time compression* as slack time is eliminated and native operations are memoized. Figure 18 shows the CPU utilization for microsoft.com over time for original and replay runs. While *total* CPU align nicely, the real site takes considerably longer than the replay, 710 ms compared to 265 ms. One potential threat due to time compression is that the lack of slack time removes opportunities for the JavaScript engine to perform service tasks such as code JITing, garbage collection, and code prefetch. This may be an important consideration in browser engineering and, as such, illustrates the inherent challenges in creating effective browser benchmarks.

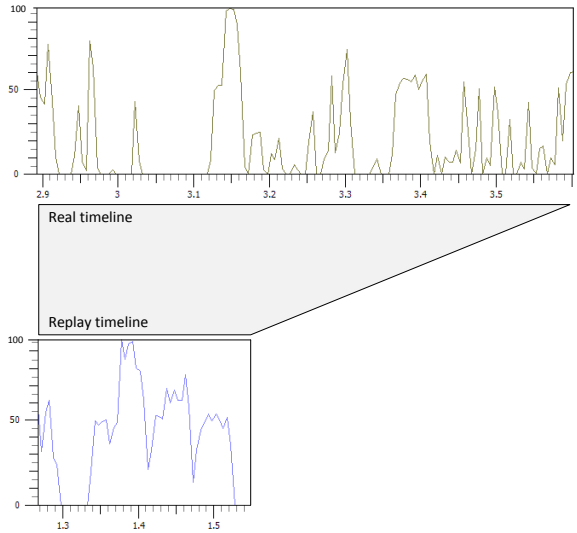


Figure 18. Time compression. CPU utilization over time. The x-axis is in milliseconds and the y-axis gives percentage of CPU used by the JavaScript engine. (microsoft.com; IE9)

5.5 JSBench vs. SunSpider

A representative benchmark should serve as a predictor of performance of a system on real sites and a guide for implementers. We have argued that industry standard benchmarks are ill suited to this task. We provide one sample experiment to back up this claim. Figure 19 gives the relative throughput improvement, over Firefox 1.5, obtained by subsequent versions when running the SunSpider industry standard benchmark and a benchmark constructed by JSBENCH from an interaction with the amazon website. The graph clearly shows that, according to SunSpider, the performance of Firefox improved over 13× between version 1.5 and version 3.6. Yet when we look at the performance improvements on amazon they are a more modest 3×. And even more interestingly, in the last two years, gains on amazon have flattened. Suggesting that some of the optimizations that work well on SunSpider do little for amazon. Note that as we have previously demonstrated [11, 13], popular sites behave rather similarly, so we anticipate the results for other large popular sites to be similar to what we are observing for amazon.

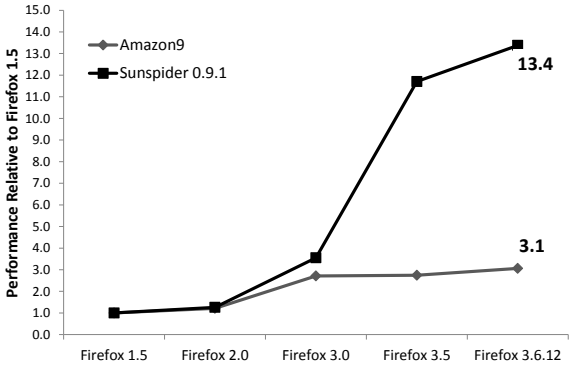


Figure 19. Speed ups. Throughput improvements of different versions of Firefox. (sunspider, amazon; FF1.5 - FF 3.6.12) Measurements on an HPZ800, dual Xeon E5630 2.53Ghz, 24GB memory, Windows 7 64-bit Enterprise. Numbers normalized to FF 1.5.

5.6 Browser-specific Replays

In this paper, our primary focus is on comparing the performance of JavaScript *engines* by running them on JavaScript-only versions of our benchmarks. However, JSBENCH does support generation of traces with some browser-specific operations left in. In these partially-deterministic modes JSBENCH does not guarantee that the program will run identically, or at all, in a different browser (because the browser may perform DOM accesses that were not encountered at recording time), but when replays can run in multiple browsers it is possible to compare the impact of other browser features on performance.

We start by looking at the performance impact of DOM operations. For this we measure the performance of a replay without mock objects for DOM reads/writes. This means

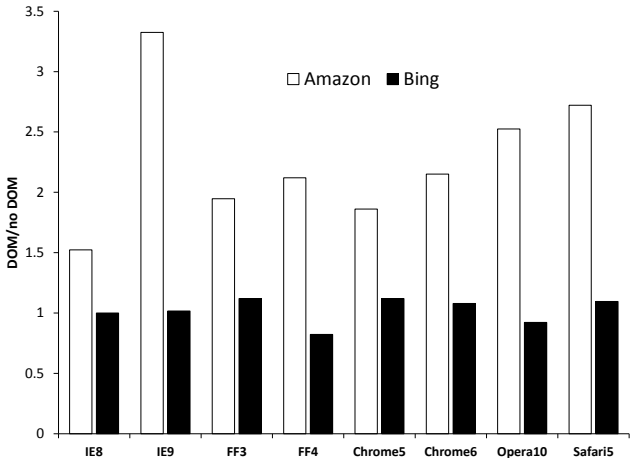


Figure 20. Cross-browser comparison. Impact of DOM operations on throughput. (number normalized to the replay without DOM; average over five runs; lower is better)

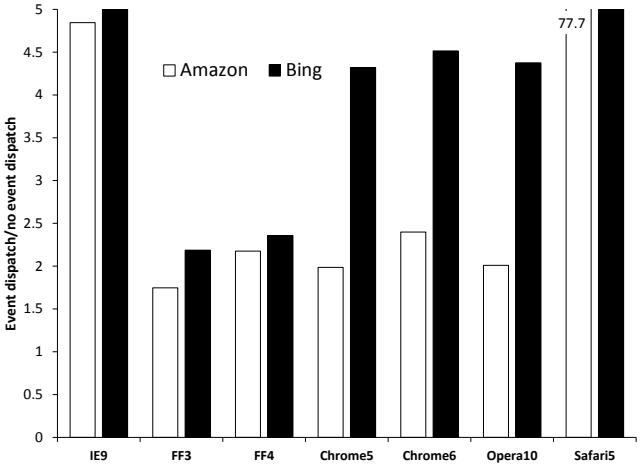
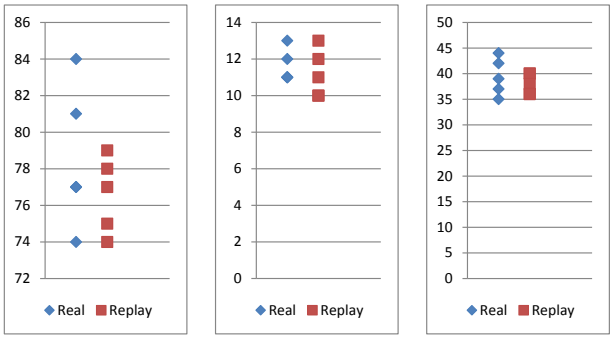


Figure 21. Cross-browser comparison. Impact of events on throughput. (number normalized to the replay without event processing; average over five runs; lower is better)

that throughput measure will include the time spent in the browser's DOM implementation. Figure 20 illustrates the relative performance of browsers with DOM turned on in the amazon and bing benchmarks. We can see that the impact of DOM operations is negligible for bing and substantial for amazon. We see that Safari5 and IE9 stand out in the case of amazon, which may be because of a slower DOM implementation or a comparatively fast JavaScript engine.

Figure 21 shows the relative cost of enabling event processing in replay as the ratio of the running times. In many browsers, the cost of event processing for the bing benchmark is relatively high, and as high as 77x in the case of Safari5. This may be because our chosen method of event dispatch through postMessage is on a particularly unoptimized code path in the case of that browser. Next, we look at how stable our execution time results are across the dif-



(a) Layout (b) Paint (c) CSS Calc.

Figure 22. Rendering and layout. Comparing internal browser operation between 5 real runs and 5 replay runs. (microsoft; IE9)

Benchmark	IE8	IE9	FF3	FF4	Chrome5	Chrome6	Opera10	Safari5
Sibelius	684	117	315	285	82	82	89	74
Amazon	342	63	104	85	110	111	147	62
Microsoft	42	4	14	11	12	10	58	5
Bing	87	11	44	51	30	27	9	9
Economist		81	103	124	48	39	40.	57
MSNBC		32	172	85	32	31	43	48
JSMIPS		7,773			4,935	3,480		12,596

Figure 23. Cross-browser running time comparison. Times are in milliseconds. An empty cell indicates that the benchmark could not produce results, due either to insufficient feature support or taking too long to execute.

ferent JavaScript engines by running each five times and looking at the standard deviation between the running times. Most browsers reliably produce consistent results. We evaluate consistency by computing the standard deviation over five runs and then normalizing it by the mean running time. For IE8, the maximum across all applications is 0.04; for FF3 it is 0.13. It is encouraging that these ratios are quite small. The browser that stands out in terms of inconsistency is Opera. For some of the benchmarks, this ratio is as high 2.1, which implies that either Opera's speed is very inconsistent, or its JavaScript time mechanisms are incorrect. In fact, the bing.com benchmark sometimes yields *negative* time on Opera, indicating that its Date s do not monotonically increase!

Figure 22 demonstrates that for the number of layout, paint, and CSS calculation events performed by the browser, the replay trace is actually more deterministic than the real trace. We can think of our replay mechanism as removing some of the inherent browser uncertainty.

Despite the fact our goal is to enable JavaScript engine comparisons, we acknowledge that our benchmarks will be used to compare browsers. We provide a snapshot of the performance of our benchmarks on browsers available at the time of writing in Figure 23.

6. Conclusions

Previous work has shown that relying on industry-standard benchmark suite leads to optimizations that do not improve the throughput of JavaScript engines running complex web applications. This paper has presented a methodology for constructing realistic benchmarks as replays of event-driven, interactive web applications. The JSBENCH tool can do so by encapsulating all the non-determinism in the execution environment, including user input, network IO, timed events, and browser-specific features. The result is a replayable program that can be deterministically executed in a JavaScript engine with a high degree of fidelity compared to the recorded trace, and high accuracy when compared to runs of the original web site. As a caveat, while we believe that we have demonstrated that creating representative benchmarks with a high degree of fidelity is possible with this approach, we do not claim that the specific benchmarks used in this paper are in fact the "correct" set to be used in ultimately comparing the performance of JavaScript engines. With JSBENCH, however, one can easily capture benchmarks that matter. As the use of client-heavy web applications evolves, approaches such as JSBENCH will enable browser manufacturers and web site builders to tailor their efforts to the ever-changing application landscape.

Acknowledgments

The authors thank Ben Livshits and Ben Zorn at Microsoft Research for their input, discussions and feedback during the development of this project as well as Steve Fink at Mozilla for providing a custom instrumented build of Mozilla Firefox for our evaluation. This material is based upon work supported by the National Science Foundation under Grant No. 1047962 and 0811631, by a SEIF grant from Microsoft Research and PhD fellowship from the Mozilla Foundation.

References

[1] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[2] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. D. Bosschere. A taxonomy of execution replay systems. In *Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[3] J. de Halleux and N. Tillmann. Moles: Tool-assisted environment isolation with closures. In *International Conference on*

Objects, Models, Components, Patterns (TOOLS), pages 253–270, 2010.

[4] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *European Conference on Object Oriented Programming, (ECOOP)*, pages 92–115, 1999.

[5] C. Dionne, M. Feeley, and J. Desbien. A taxonomy of distributed debuggers based on execution replay. pages 203–214, 1996.

[6] A. Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 465–478, 2009.

[7] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Symposium on Operating Systems Principles (SOSP)*, pages 17–30, 2007.

[8] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound Web 2.0 applications. In *Conference on Foundations of Sofware Engineering (FSE)*, pages 350–360, 2008.

[9] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for JavaScript applications. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[10] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for Web 2.0 applications robustness testing. In *International Symposium on Software Reliability Engineering (ISRE)*, 2010.

[11] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Conference on Web Application Development*, 2010.

[12] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[13] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[14] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing Web 2.0 applications through replicated execution. In *Conference on Computer and Communications Security (CCS)*, pages 173–186, 2009.