Imagen: Runtime Migration of Browser Sessions for JavaScript Web Applications

James Lo
Department of Computer Science
University of British Columbia
Vancouver, Canada
tklo@cs.ubc.ca

Eric Wohlstadter
Department of Computer Science
University of British Columbia
Vancouver, Canada
wohlstad@cs.ubc.ca

Ali Mesbah
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

ABSTRACT

Due to the increasing complexity of web applications and emerging HTML5 standards, a large amount of runtime state is created and managed in the user's browser. While such complexity is desirable for user experience, it makes it hard for developers to implement mechanisms that provide users ubiquitous access to the data they create during application use. This paper presents our research into browser session migration for JavaScript-based web applications. Session migration is the act of transferring a session between browsers at runtime. Without burden to developers, our system allows users to create a snapshot image that captures all runtime state needed to resume the session elsewhere. Our system works completely in the JavaScript layer and thus snapshots can be transfered between different browser vendors and hardware devices. We report on performance metrics of the system using five applications, four different browsers, and three different devices.

Categories and Subject Descriptors

D.3.2 [**Software**]: Programming Languages—*JavaScript*; E.2 [**Data**]: Data Storage Representations—*Object Representation*

Keywords

JavaScript, session migration, HTML5, JSON, DOM

1. INTRODUCTION

The World Wide Web was originally designed around the notion of uniquely identifiable resources. Using the URL of a resource, the user could point to and load a specific state of a website into their browser [15]. This simple stateless client/server interaction contributed to the success of the Web. However, due to increasing complexity of web applications (referred to as *apps*), considerable effort on the part of developers is now required to achieve state persistence.

With the evolution of web technologies, browsers, and HTML5 [5] a great deal of application state is being offloaded to the client-side. In order to achieve more responsive apps, JavaScript is increasingly used to incrementally mutate the Document Object Model (DOM) in the browser to represent a state change, without requiring a URL change. Additionally, with new HTML5 APIs apps can feature advanced graphics, animation, audio, and video. Therefore, capturing and migrating a particular state of an app is not as simple as saving and loading a URL any longer. It requires developers to manually implement code for persisting the transient browser state (i.e. state that normally would be lost once a user closes a browser tab). While some libraries and APIs [4, 7, 30] provide support for object persistence, developers are still obliged to register and track individual objects programmatically, which can be tedius and error-prone. Since persistence is well-known to be a crosscutting concern [34, 32], adding it to existing code is difficult because it requires changes scattered across various modules. Furthermore, such libraries only support persistence of simple JavaScript objects, and not other application state such as function closures, event-handlers, or HTML5 media objects.

In this paper, we investigate the use of *session migration* to address this problem. Session migration is the act of transferring a session between browsers, possibly on different platforms, at runtime. We propose a novel technique and tool, called IMAGEN[1], for migrating client-side session state of web apps across different browsers and devices. Our technique enables end-users to seamlessly capture the runtime client-side browser state at a desired instance, and later restore that state in a different browser and continue using the app from there.

While there is some previous work on Web app migration [3], that system only supports migration of traditional data-structures and not full application runtime state which include Javascript function closures, event-handlers, and HTML5 media objects. Thus that approach is not applicable to the type of applications supported and evaluated in this research. Also, previous work [27, 2] on event-logging of JavaScript could theoretically be applied for session migration. However, that work was intended for the purpose of development-time debugging, and we show in our evaluation that it is not practical for end-user session migration.

IMAGEN works through a combination of novel JavaScript transformations. Such transformations can be applied in two different ways: developer-initiated or user-initiated. The developer-initiated transformation is applied by a software developer to their code prior to application deployment. Alternatively, end-users can enable the transformation themselves by using a provided transformation HTTP proxy. Either way, no extra coding is required to achieve migration of sessions.

[1] IMAGEN means *image* in Spanish.

WWW 2013, May 13-17, 2013, Rio de Janeiro, Brazil
ACM 978-1-4503-2035-1/13/05.

This paper makes the following main contributions:

- We propose an automated generic and transparent approach for persisting and migrating the transient session states of JavaScript web apps;
- We illustrate how entire client-side states, including those in JavaScript function closures, event-handlers, and HTML5 media objects can be captured and serialized;
- We describe how the serialized session state can be brought back to life in a different browser;
- We present the implementation of our approach in a tool called IMAGEN[2]; an online video[3] provides a demonstration.
- We report the efficiency of our approach through an empirical evaluation using five existing apps. Our results indicate that IMAGEN adds less than 10% execution overhead to all our test apps and less than 2% in most cases.

2. MOTIVATING EXAMPLES

Robots Are People Too [35] (RAPT) is a side-scrolling two player platform game. The game features different challenges, drones, and rewards. It invites gamers to invest sufficient time and effort to finish levels and make progress. A gamer may want to persist or migrate her session of gameplay for any of the following reasons:

- *Strategy*: She perceives a risky move ahead and wants to seamlessly try again when it fails without repeating previous effort.
- *Time*: She has to work on something else and wants to close this game completely from the browser. A reason could be to free up some system resources since the game can be performance intensive.
- *Location*: She is going somewhere and wants to continue the game at another location or on another device.

The current version of RAPT does not provide a feature for saving progress during a game. By using IMAGEN, end users can persist and migrate such game state without requiring developers to provide any additional coding.

SketchPad [13] is a painting app that makes use of the HTML5 `<canvas>` tag. Once the painter has chosen to save, the image gets saved to the Portable Network Graphics format (PNG); however there is no option in the app to load a PNG later for further editing. By saving their browser session using IMAGEN, a user can resume editing some picture at any later time. IMAGEN also remembers and migrates all the many configurable settings that SketchPad has, e.g. the color palette, gradient, pattern, and tools settings. In the current version of SketchPad all of these settings are lost when a user closes their browser.

3. IMAGEN DESIGN

We start by presenting an overview of some challenges in session migration (Section 3.1), followed by a high-level architectural overview of the components involved in our approach (Section 3.2).

[2]Source code is available at: `http://www.cs.ubc.ca/~wohlstad/imagen.html`

[3]Video is available at: `http://www.cs.ubc.ca/~wohlstad/imagenVideo.html`

```
1 //Attempt (and fail) to serialize the
2 //user's session by JSONizing 'window'
3 var snapshot = JSON.stringify(window);
4
5 //Attempt (and fail) to unserialize a
6 //user's session by assigning
7 //parsed string to 'window'
8 window = JSON.parse(snapshot);
```

Sample Code 1: Essence of Snapshot Imaging. This code fails horribly on regular apps but becomes possible using Imagen.

3.1 Challenges

In order to explain the technical challenges for session migration, we start from an incorrect strawman implementation of saving/loading a snapshot image of some browser session (shown in Sample Code 1). In JavaScript, the global variable `window` provides a context from which both native browser APIs (such as the DOM) and application-specific state (in the form of JavaScript objects) can be accessed by programmers. JSON [22] is the popular serialization format of JavaScript and can be used to serialize (`JSON.stringify`) and unserialize objects (`JSON.parse`). Thus it would seem reasonable that to migrate a user's session, one might be able to simply `stringify` the whole `window` object (line 3). Ideally, this would return a string capturing all runtime state needed for migration. Then later, on another browser, the stringified snapshot could be parsed back into `window` (line 8) and the user's session would resume. Unfortunately, this will not work in practice.

Capturing a snapshot for migration is much more challenging, for a number of reasons:

1. *Function Closures.* In addition to objects, JavaScript state includes function instances called *function closures*. This kind of function/object hybrid is not easy to serialize.
2. *Event-handler state.* Event-handlers are the driving force of execution in JavaScript. They create a schedule of activity that is not supported by existing serialization mechanisms.
3. *HTML5 rich-media objects.* Modern web applications make use of rich-media objects from the HTML5 standard which have unique serialization requirements.

All of these problems need to be solved without introducing burden on the developer or end-user, in particular IMAGEN should be:

1. *Generic and Interoperable*: End users should be able to migrate a variety of apps and should have the freedom to use a snapshot in any device of their choice.
2. *Automatic*: Enabling session migration should not require additional coding for developers and only minor setup configuration.
3. *Efficient and Scalable*: End users should experience the same level of interactivity as the original app. This means IMAGEN's overhead to the app's execution should be minimal.

3.2 Architectural Overview

We describe the components involved in the migration of a running app, as depicted in Figure 1. The figure is divided into two: the top half (*Save Snapshot Flow*), the bottom half

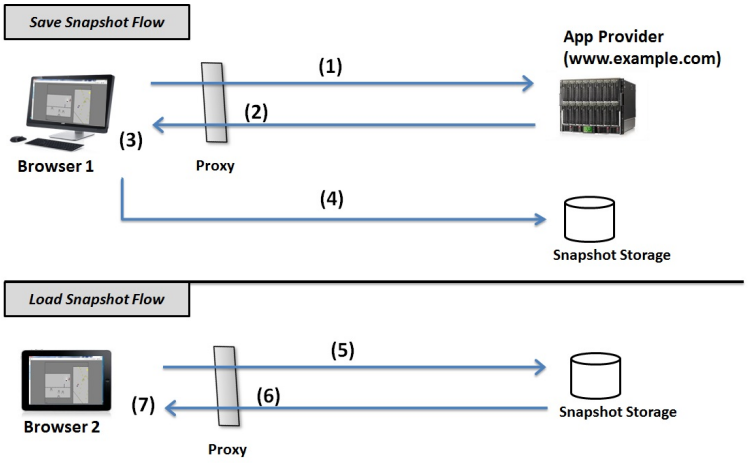

Figure 1: Imagen Architecture: (top) Starting up an app and saving a snapshot; (bottom) Loading a previously saved snapshot to a different device.

(*Load Snapshot Flow*). First, a user starts to load an app in their browser and execute it as usual (1); e.g. entering the URL or navigating from a search page. In order to make migration possible, the JavaScript source code of an application must be transformed and instrumented with additional code. This can be done by the developers using a source code processor, prior to deployment. Alternatively, instrumentation can be transparently injected by an end-user by making use of a provided HTTP proxy which runs on the user's own machine (shown as (2) in the figure). Our technique supports both, and either way, no manual changes to the application code is necessary. Transparency is important because developers often mash-up 3rd party APIs into their Web application. For example, RAPT embeds Analytics and the minified version of jQuery hosted by both Google.

Ideally, there should be no noticeable change in the app's behavior after instrumentation. Using a simple GUI button (added to the bottom of each web page by the instrumentation) a user can take a snapshot at any point during execution (3). This snapshot is then saved to a secondary storage (4), either on a remote web service (*Snapshot Storage* in the figure), or on the user's local drive. Either way, the user is provided with a URL, which can be used to retrieve and load the snapshot.

Sometime later, the user opens a new browser, which can be on a different device. In the example figure, the user migrates the application from their desktop to a tablet (e.g., iPad). The user then enters the previously given URL in the new browser (5). If instrumentation was provided by the application developer, step (6) is not necessary. Otherwise at (6), the proxy redirects the user's browser to the original URL where the snapshot was taken. However, rather than returning the content at that URL, it returns the saved snapshot instead. This step allows the restored app to run in the same browser security domain as the original application[4]. After the app is loaded into the new browser, it seamlessly continues exactly where it had left off (7).

4. TECHNICAL DETAILS

[4]We assume the user trusts the proxy so no new security threats are created by this technique.

We continue to describe the challenges laid out in Section 3.1 and the solutions we implemented.

4.1 Migrating Function Closures

The basic idea for capturing an app's execution state is to traverse and serialize its object graph [20]. However, certain "edges" in this graph cannot be traversed by the application itself and are kept as internal browser state. These "edges" correspond to the links between functions and *closure-bound variables*. In this subsection, we describe the problem of closure-bound variables and our solution through an example. We have made this example as simple as possible, just to focus on the essential problem created by closures.

4.1.1 Example of Closure Usage

In JavaScript, functions act as both executable code and objects. Like objects, functions can have properties assigned to them and they can be assigned as values to properties of other objects. Also, there can be many instances of the same function type. Whereas in static languages such as Java, we may assume that functions exist when a program first starts, in JavaScript, function instances are created dynamically when the statement where they are defined is executed. At the moment when they are created, they also become associated with certain variables that were in scope when they were created. This is referred to as creating a function closure [1] (i.e. closure). While JavaScript has been criticized for some poor language design decisions, closures have been recognized as one of its good qualities [11].

Sample Code 2 presents a simple counter using a function closure. On line 1, a function is defined called `CreateCounter`, which takes as an argument a `count_val` to be used as the counter's initial value. On line 2, a new object is created for the counter, and on line 3 a function is added to the object. This function (lines 3-6) takes an argument to increment and then prints the counter to the screen. The counter state is captured as a closure-bound variable `count_val` simply by referring to the name of the argument passed to `CreateCounter` (on line 1).

Closure-bound variables allow programmers to create references between: (a) variables in a created function and (b)

```
1  function CreateCounter(count_val) {
2    var count = new Object();
3    count.inc = function(add) {
4      count_val += add;
5      alert(count_val);
6    };
7    return count;
8  }
9
10 //Example usage
11 window.myCounter = CreateCounter(10);
12 window.myCounter.inc(5); //prints '15'
```

Sample Code 2: Example use of closures.

```
1  function CreateCounter(count_val) {
2    var CreateCounterScope = new Object();
3    CreateCounterScope.count_val = count_val;
4    var count = new Object();
5    count.inc = function(add) {
6      count_val += add;
7      CreateCounterScope.count_val = count_val;
8      alert(count_val);
9    };
10   count.inc.parentScope = CreateCounterScope;
11   return count;
12 }
```

Sample Code 3: Example of closure instrumentation for CreateCounter.

objects that are referenced in the scope that the function is created. For example, in Sample Code 2 we see (a) `inc` being created on line 3 with the closure-bound variable `count_val` on line 4 and 5. These references on line 4 and 5 refer to (b) the object passed as parameter `count_val` on line 1. This link is simply achieved by using the same variable name in the created function as some variable that appears in the scope it is created.

Closure-bound variables are different from local variables because they exist for the lifetime of a function object, not simply during one execution of the function. For example, when some execution of `CreateCounter` goes out of scope, the created function `inc` will still refer to the object that had been passed as the parameter to `CreateCounter`.

Suppose the example code in Sample Code 2 on lines 11-12 is executed. At this point a counter object is easily accessible through `window.myCounter`. Furthermore, the `inc` function can be accessed as `window.myCounter.inc`. However, interestingly, it is not possible to access the counter's actual state (referred to as `count_val`). Thus there is no straightforward way to serialize this counter so that it can migrate or persist for another browser session. It is important to note that this is true for any closure-bound variable and is not specific to this example.

To solve this problem we instrument JavaScript code to monitor the use of closures in an application so that we can reliably serialize and unserialize execution. We describe this solution in five parts: (i) explicit scope, (ii) monitoring changes to closure variables, (iii) associating scopes with functions, (iv) closure serialization, and (v) closure loading.

4.1.2 Closure Saving

```
1  function CreateCounterScope_1() {
2    var count_val = 15;
3
4    objIndex[ID].inc = function(add)
5    {
6      count_val += add;
7      alert(count_val);
8    };
9  }
```

Sample Code 4: Example of generated code for closure restoration after migration.

First, to deal with (i), we make closure-bound variables available to our system by instrumenting the code. This is done to explicitly keep track of a scope object for each function execution which has variables that will be referenced by closures. This is demonstrated in Sample Code 3. Our instrumentation code is shown in *italic*. On line 2 we create an object that will track the scope of `CreateCounter` executions. Notice that for the function `count.inc` there is no corresponding explicit scope object. This is because that function has no variables that will be referenced by closures (this is clear because it has no functions defined in its scope). Once a scope object is created, all of the variables that will be referenced by closures are added to the object, so we can keep track of their values. This occurs for example on line 3, where the variable `count_val` is added to the newly created scope object.

Second, to monitor changes to closure variables (ii), whenever a variable that has been added to an explicit scope object is assigned to, we assign the same value to the property of the scope object which is responsible for monitoring that variable. This is shown on line 7. When `count_val` is incremented by `add` we change the explicit scope object to the updated value. This ensures that the values in the explicit scope object are always an accurate reflection of the actual program state.

Third, at this point we have a "mirror" of the closure-related program state. Essentially, by keeping a copy of this state in the application, we now have access to it for serialization. However we still need to keep track of which scope objects are used by which function objects. To assign our scope objects to functions (iii), we attach the explicit scope objects directly to the functions that use them. Since function definitions form a tree-like hierarchy, each function is matched to the scope object of its parent function (immediately enclosing function). This is shown in line 10, immediately after the function `count.inc` is created, we bind the explicit scope object of its parent to the function with a special property (`parentScope`).

Fourth, now that we have a data-structure modeling the closure-bound variables, their values, and the relationships between the function scopes, dealing with serialization of closures (iv) is straightforward. Whenever a function object is encountered by our serializer, the following information is serialized: the name of the function and the hierarchy of explicit scope objects reachable through the special `parentScope` property.

4.1.3 Closure Loading

Finally, this serialized closure information needs to be restored (v), possibly on another browser. However, it is

not trivial to recreate the binding between each closure-bound variable and the value it had at the time of serialization. For example, consider the example code at the end of Sample Code 2. Imagine we take a snapshot of this program after these two lines execute, in browser A, and then restore it on another browser B. Now if a call to `window.myCounter.inc(2)` is made in B, it must respond with an alert of '17'. But how do we get the variable `count_val` in the restored version of `window.myCounter.inc` to reference its properly restored value of `15`, which was taken at browser A? Or in other words, how does a generic tool-driven transformation provide this without understanding the semantics of the program?

A naive solution would be for our tool to call the function `CreateCounter` passing an argument of `15`. In this particular case, it would create a new counter object with the initial value 15. However, a tool could not discover these semantics of the program in the general case because it would require a sound and complete static data-flow analysis, which is known to be undecidable [24, 29]. Furthermore, if `CreateCounter` had side effects, these side-effects would be triggered upon session migration, which would violate the original program semantics. For example, if the function `CreateCounter` issued an `XMLHttpRequest` (XHR) request, then calling the function for the purpose of restoring a previous program state would cause the XHR to be duplicated.

Instead, our solution involves generating new code that causes closure-bound variables in functions to become bound to their properly restored values. We unserialize the explicit scope object for the execution of a function by generating a new function, as shown in Sample Code 4.3.1. We generate one variable declaration for each value saved in the scope object. For example, on line 2, the `count_val` variable is defined and assigned to its current value. Recall that this code is generated so we can insert whatever values are necessary into the text of the code to initialize variables. In this case, `count_val` is assigned its current value of "15".

When a snapshot is restored, this code is evaluated by the target browser, recreating the closures and binding their closure-bound variables to the proper values. We can see this in lines 4-8. We keep a map called `objIndex`, which contains all objects that are being restored during deserialization. This map is indexed by a unique `ID` assigned to each unique object, because there can be multiple references to the same object, including self- or circular- references. In our example, `objIndex[ID]` points to our counter object. In order to restore the function property `inc` for our example counter, the function definition for `inc` is generated inside the scope of our generated function that models its parent scope `CreateCounterScope_1`. Now the new copy of `inc` has its `count_val` variable bound to the appropriate value of '15' using the standard mechanism for binding closure-bound variables. If a call to this counter is made now with an argument of '2', it will appropriately respond with '17'.

Our index is extensible so that developers can define how each object type is (de-)serialized. For example, we extended our index to cover native objects such as Dates and Arrays.

4.2 Event-Handlers

In order for a web app's state to migrate, we need to capture and restore its *event state*. To explain this challenge and our solution, we first start with a brief explanation of the salient details of the JavaScript event-model.

4.2.1 Relevant JavaScript Event Basics

Browser embedded JavaScript provides a single-threaded concurrency model for handling the browser's user interface. Programmers register event-handler functions with some event type that should trigger them, e.g. when a button is clicked by the mouse. When such an event occurs, any corresponding event-handlers become queued for execution. Since handling of events is single-threaded, event-handlers may not actually begin execution immediately when their corresponding event is triggered. If another event-handler is executing at that time, other event-handlers are placed on an *active queue* and are serviced by the browser generally using a first-in-first-out (FIFO) policy.

In order to monitor event handler registration, we instrument the APIs that are used to register events (e.g. `addEventListener`). Additionally, when each handler function is registered, we wrap that function with a Decorator [17], that records when the handler execution begins and when it completes. This is needed so we can monitor which events have completed and which are still pending (further details described below). Since our transformation for closures allows functions to be included in a snapshot it is easy to include each event-handler function as well. However, some types of events (especially timer events) require special treatment, so we divide them into three categories and describe the differences below.

4.2.2 Event Categories

Based on the HTML5 specification, we divide event types into three categories with differing behavior from the perspective of migration.

UI Events: When a relevant event occurs on a given UI element, any corresponding handler function is added to the end of the active queue. UI Events do not require any technical solution because of the FIFO policy of browsers. When Imagen is asked to save a snapshot by the user, this request is added to the end of the active queue, so any relevant event handlers already queued because of previous UI interactions are guaranteed to execute before the snapshot is taken. When restoring a snapshot, we intercept all events that can have side-effects: e.g. onload.

Asynchronous I/O Events: JavaScript uses an asynchronous model for handling I/O requests that could potentially have high latency (such as XHR). For this purpose JavaScript requires a callback function to be registered to receive a result from an I/O request. Results which are pending at the time of a snapshot can pose a challenge. For this reason, we capture all requests for I/O in our instrumentation of JavaScript. If a snapshot is taken when a response for some request has not yet occurred, we record the request in the snapshot. Later, when a snapshot is loaded, we replay such requests so that the result can arrive at the migrated location.

Timer Events: Event-handlers can be registered to start executing after some given time has elapsed. We refer to the time that a handler is registered at as t_{reg} and the time to elapse as $interval$. Such handlers come in two flavors, (1) `setTimeout` causes a handler to execute one time only. At time $t_{reg} + interval$, the browser will add the corresponding handler function to the active queue; (2) `setInterval` causes a handler to execute periodically for a given interval. Here,

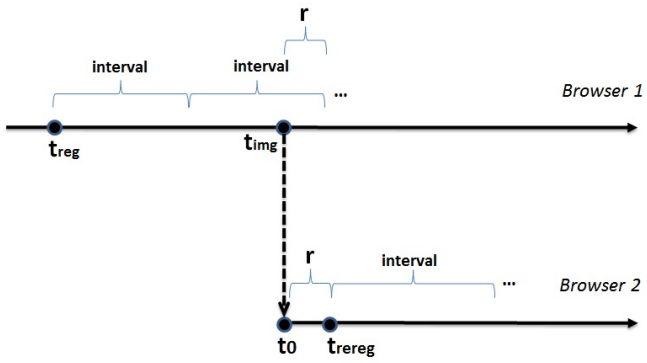

Figure 2: `setInterval` example. Time flows from left to right along the axis (top) at Browser 1, and (bottom) at Browser 2.

the browser will add the handler function at time $t_{reg} + interval*x$ for all integers $x > 0$. If any timers are scheduled to execute when a snapshot is requested, this schedule needs to be saved. Dealing with timers is more complicated than for the other two categories, so we describe our solution in more detail below.

4.2.3 Saving and Loading for Timers

We describe here how we handle the case of `setInterval` timers as `setTimeout` is a simpler case of this general behavior. As in the illustration of Figure 2, suppose a script registers a `setInterval` in some browser (Browser 1) at some time, t_{reg}. Let the time of the interval registered be *interval*. Now suppose the user requests a snapshot at some time, t_{img}, where $t_{img} > t_{reg}$. This scenario is illustrated on the top timeline of the figure.

At this point, the registered event-handler needs to be activated at some time in the future. Since the future time may occur when the app has migrated to another browser, IMAGEN must handle this case. To handle this, we record the residual time, $((t_{reg} - t_{img}) \bmod interval)$, in the snapshot, labeled as r in the figure. This represents the time remaining until the handler should be activated.

Later, suppose a snapshot is transferred to another browser (Browser 2), depicted by the dotted line in the figure. At this time, t_0, IMAGEN automatically re-registers a `setTimeout` handler in the new browser but setting the timeout to the recorded residual value. This will cause this shorter interval to activate only once after restoration. After the handler has executed once on the short interval, it then needs to resume on the original interval at time t_{rereg}. To achieve this, we use our injected decorator code to re-register the original `setInterval` immediately after the handler has executed once on the short residual interval. Now the interval will continue executing seamlessly without unexpected changes in runtime behavior.

4.3 HTML5

A traditional JavaScript object is simply a set of key-value pairs (i.e. a map or dictionary). However, native browser objects such as those commonly provided by the newer HTML5 APIs include some objects which have differing behavior from traditional objects. In our research, we first implemented our solutions to the problems of function closures and event-handlers described in Sections 4.1 and 4.2. We then attempted to use our prototype on existing web sites. This trial uncovered two new problems related to HTML5. These observed problems are related to "opaque" objects and stream resources.

```
1  img = document.createElement("img");
2  img.src = "http://example.com/img.jpg";
3  pattern = canvas.createPattern(img);
```

Sample Code 5: Example of CanvasPattern (an opaque HTML5 object).

4.3.1 Opaque Objects

Traditionally, the entire state of an object can be accessed through its properties. So including the state of an object in a snapshot simply requires enumerating property values (i.e. as would be done by JSON). However, some HTML5 objects include state which is not accessible through its properties. These objects are referred to as "opaque" in HTML5 specifications (e.g. [8]).

An example is illustrated in Sample Code 5 with an HTML5 `CanvasPattern` object. On line 1, an HTML `<img>` is created and the source URL is set on line 2. On line 3, an image pattern is created from this image. Such patterns are used, for example, to create a tiled background from a single image tile. The function call on line 3 yields our result assigned to the `pattern` variable.

If we are to save a snapshot for an app which includes the pattern, we need to know which image URL is to be used by the pattern. Unfortunately, although the API uses an image to construct the pattern, there is no corresponding "getter" in the API to interrogate which image was responsible for the pattern. In other words, we would like to use something like `pattern.img` to determine this association since this is how traditional JavaScript objects maintain properties. However, in this case such state simply becomes hidden to JavaScript behind the native implementation of the browser, i.e. `pattern` is opaque and hides its internal structure.

To deal with this problem, our instrumentation intercepts HTML5 API functions which construct or mutate opaque state. We record both the function name being called and all of its arguments. We associate these records with the opaque object being created or mutated. This yields a log associated with the opaque object in our snapshot. Later, when we load a snapshot, to recreate the object state, we replay this log of function calls.

Note that theoretically, we could use the same log replay approach for all objects (not just the opaque HTML5 objects). However, this would yield a log size which is impractical. We evaluate this further in

4.3.2 Stream Resources

Similar to "opaque" objects, streaming media objects, such as `<audio>` and `<video>` objects, do not behave like traditional objects. This is because the state of the object consists of a data stream which is progressively buffered over time. Such buffers are generally too large to include in a snapshot, so it is more practical to reload them from their original location after migration. However, since loading of

these objects is an asynchronous operation, resuming execution of a snapshot that includes them creates difficulties synchronizing their playback with the script execution.

An example of this problem occurred when we applied one of our earlier prototypes to the *ColorPiano* app [12]. ColorPiano is a piano teaching animation. As a song is played, the app animates a slider of sheet-music notes and animates keys playing each note. Session migration would be useful here as it allows piano students to pause and resume at any particular positions in any songs they are practicing. ColorPiano does not include this feature in its current implementation. However, when we loaded a saved snapshot of ColorPiano, we noticed that the animation and the audio became out of synch. This is because upon loading the snapshot in a new browser, the audio stream was not yet ready to play at the position of the song where we had left off. The problem is clearly not specific to ColorPiano but is general for any applications using Imagen with such media objects.

To deal with this, we make use of support for *random seeking* in HTML5 and modern media servers. This allows scripts to initiate buffering at any position in a stream. When a snapshot is saved, we record the stream position of each media object and its status (whether it is *playing* or *paused*). Then when a snapshot is loaded we request that playback for these objects resume at the position recorded in the snapshot. We are able to use a function in the HTML5 specification called `seekable` to determine when the buffer for these objects is playable at the specified position. Once all the playing objects are playable at their previously recorded snapshot position, we allow the original execution to resume, as it is now synchronized with media. Because we control the seeking, we intercept events triggered by our code and prevent them from causing side effects.

For example, in ColorPiano, this creates an additional delay of a few seconds when loading a snapshot (specific data provided in the evaluation). However, once the snapshot resumes, the sheet music animation and the corresponding audio resume seamlessly.

4.4 Other implementation details

Our implementation consists of two main parts: a source code transformation for JavaScript (written in Java) and a library of JavaScript functions.

The Java-based transformer is built on top of Mozilla's Rhino open-source project. It provides us JavaScript in the form of abstract syntax trees (ASTs) which we then analyze and transform with our own code. Our own Java code is 6,923 lines. The transformer can be run by a developer through the command-line to transform their JavaScript code. Alternatively, the transformer can be hosted in an HTTP proxy by any user. We added a plugin to the WebScarab [36] proxy for this purpose. In the future we plan to implement this proxy feature as a browser plugin also.

Our JavaScript library is injected into a web app by the transformer (by inserting a `<script>` tag into the DOM). The library performs most of the work of saving and loading snapshots. It retrieves information that was stashed away by the instrumentation. This information is combined with other data available through `window` to build a JSON formatted snapshot file consisting of: function closures, plain JavaScript objects, event-handlers, media objects, the DOM, and optionally any cookies for the web app. To JSONize the DOM we make use an existing library called JsonML [23]. Our own JavaScript library is 5,028 lines of code.

Cookies can be included so the same app can continue running without interruption on another browser even when cookie data is used by the app. Since users are in control of their own snapshot data, this does not leak the user's cookies to any third-party. However, since some users may want to share their snapshot with others, this feature is optional. In our experience, if cookies are required by the app, but they are stripped from the snapshot, the app will treat the user as though their session had timed out. In that case, a user would simply need to provide any credentials (such as username and password) another time when they load a snapshot.

5. EVALUATION

We evaluated Imagen from the perspective of a number of research questions:

1. Is it possible to serialize a snapshot of the execution state of a running JavaScript app and resume execution on another browser? This is evaluated by using five different apps from three domains: gaming, data visualization and multimedia.
2. Is the performance of Snapshot efficient enough for practical use? This is evaluated through a number of performance metrics divided into three categories: Instrumentation (Section 5.1), Execution Overhead (5.2), and Snapshot Lifecycle (5.3).
3. To what extent can this work for different browsers, and devices? This is evaluated by performance metrics for one app across four browsers and three devices.
4. For the purpose of session migration, how does the approach of Imagen compare to MugShot's approach of event logging [27]? This is evaluated by a comparison of snapshot (or log) size for each approach (Section 5.4).

Evaluation was performed on a number of different browsers and devices. Unless otherwise stated, measurements come from Google Chrome v.23, running on a Windows 7 quad-core 2.8GHz laptop with 8GB of RAM.

We evaluated metrics across five HTML5 apps:

1. Robots are people too (RAPT): A side-scrolling game as described in Section 2.
2. Convergence: A puzzle strategy game [9].
3. SketchPad: An image editor as described in Section 2.
4. Genoverse: A genome browser [18]. Session migration is important for Genoverse because of the magnitude of data that users need to navigate. While Genoverse allows users to tag certain locations in a gene sequence for reference in navigating data, such tags are not saved across sessions. Imagen allows tags and other session-specific data to be saved in Genoverse without any additional programming labor.
5. ColorPiano: A piano teaching animation program as described in Section 4.3.2.

5.1 Instrumentation Overhead

Figure 3 shows several metrics related to our JavaScript instrumentation. This occurs at item (2) in the Save Snapshot Flow (Figure 1). The table includes columns for: the application tested, the aggregate size of JavaScript files, the increase in size due to our instrumentation, and the time of instrumentation. From the third column (JS Size Increase)

App Name	JS Size Original (kB)	JS Size Increase (%)	Instrumentation Time (ms)
RAPT	225	24%	702
Convergence	381	18%	780
SketchPad	200	13%	566
Genoverse	349	14%	801
ColorPiano	236	16%	636

Figure 3: Instrumentation metrics for a number of apps.

App Name	Browser Type	Device Type	IMAGEN Exec. Overhead (%)
RAPT	Chrome	PC	1.4%
RAPT	Firefox 17	PC	3.8%
RAPT	IE 9	PC	9.5%
RAPT	Safari 6	Mac	2.9%
RAPT	Safari 5.1.1	iPad 2	8.0%

Figure 4: Execution Overhead for RAPT across a number of browsers and devices.

we can see that our instrumentation adds less than 25% to the aggregate file size in all cases. We could reduce this size somewhat in the future if we implemented JavaScript minification. The number varies between applications because the instrumentation depends on which language features or APIs are used. In the fourth column, we see time taken by our proxy to instrument JavaScript files. In the case where JavaScript is transformed on-the-fly this extra time is likely to be noticeable by the end-user, however, it was less than one second in all cases. For cases where a developer applies our transformation offline, this instrumentation time is not perceived by the end-user.

5.2 Execution Overhead

Since our approach requires JavaScript code to be instrumented, it will have some overhead during execution. This metric measures that overhead by profiling applications with and without our instrumentation. This corresponds to overhead incurred at steps (3) or (7) in Save Snapshot Flow or Load Snapshot Flow. Execution times are measured by wrapping each event handler execution with a Decorator that records clock time before and after the handler. These elapsed times for handler execution are aggregated over one minute of program activity to record the time spent by the browser executing JavaScript. These measurements are averaged over 10 trials, then compared for the case of instrumented and uninstrumented versions of the app.

The execution overhead could be affected by both: (i) the browser used to host the application or (ii) the particular application which is running. This yields three-dimensions of data (metric, browser, and application). However, while it is interesting to see how the metrics are affected for each browser or each application, there is not much useful information to be gleaned for each combination of browser and application together. So we present the data in two different two-dimensional tables. First, for the case of varied browsers and device[5] (Table 4) with the application chosen as RAPT. Then, second, we present the data for the case of varied applications (Table 5), with the browser chosen as Chrome.

In Figure 4 we see the execution overhead for RAPT in the fourth column. We performed the same test case for each row[6]. Clearly, the device and browser has an effect on this overhead, but in all cases it was less than 10%. The reason this number is fairly low is because, although features like closures, events, and HTML5, are important, the majority of JavaScript statements are not using these features directly. Our instrumentation only needs to be added for those statements directly using these features. Since the overhead was very small for Chrome on our PC laptop, we hope that this level of overhead could be achieved for the other platforms through further optimization. We chose RAPT to evaluate on several platforms as it was the most CPU intensive app in our group.

Figure 5 shows the overhead for the other apps on our PC using Chrome. Here the overhead is even lower which is likely due to the fact that they simply use less of those aforementioned features, per line of JavaScript, than RAPT. To ensure the same test case is used for the instrumented and uninstrumented version, we use a UI recorder/playback program [21]. For all 5 apps we we verified successfully through simple inspection that the program execution resumed after migration without errors.

App Name	IMAGEN Exec. Overhead (%)
Convergence	0.97%
SketchPad	0.21%
Genoverse	0.43%
ColorPiano	1.2%

Figure 5: Execution Overhead for apps on PC Chrome.

5.3 Snapshot Lifecycle

The metrics related to the snapshot lifecycle are:

- Snapshot Save Time: The time it takes to create a serialized snapshot of the running application's execution state, in milliseconds. This occurs immediately before step (4) in the Save Snapshot flow.
- Snapshot Load Time: The time it takes to recreate a running application from a given serialized snapshot of the execution state, in milliseconds. This takes place immediately before step (7) in the Load Snapshot Flow.
- Snapshot size: The size of the serialized snapshot which would need to be transferred over the network for migrating between devices, in megabytes. This is the amount of data which needs to be transferred during step (4) or (6) in either Save Snapshot or Load Snapshot Flow.

In Figure 6 data is provided for the time to save and load a snapshot for each of the apps. The first bar in each column shows the save time, with the second bar showing the load

[5]The Mac is a 2.2GHZ i7 with 8GB RAM running OS X 10.7.3

[6]We simply move the character in one direction continuously since this is easily repeatable.

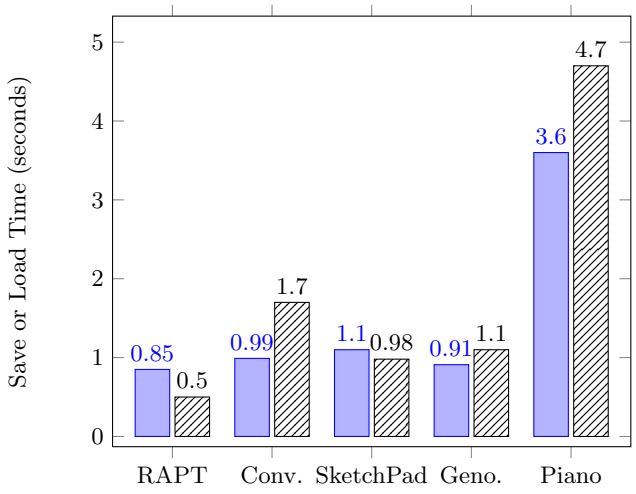

Figure 6: Snapshot Save Time (solid) and Load Time (striped) for: RAPT, Convergence (Conv.), SketchPad, Genoverse (Geno.), and ColorPiano (Piano).

time. Unsurprisingly the time to save and the time to load appear to be correlated, for example, ColorPiano has both the highest save and load time. Examining Figure 7 we see this correlation carries over somewhat to the snapshot size, although not precisely. On one hand, it makes sense that ColorPiano has both the highest save time and the largest snapshot size. On the other hand, RAPT has the fastest save-time but not the smallest snapshot size. This is likely due to the fact that RAPT carries most of its state in one large `canvas` object so there are few objects in RAPT for our tool to process, whereas other apps have their state spread across many more fine-grained objects which each require separate processing.

App Name	Snapshot Size (MB)
RAPT	2.037
Convergence	2.417
SketchPad	1.953
Genoverse	1.379
ColorPiano	8.329

Figure 7: Snapshot Size for a number of apps.

Figure 7 shows the uncompressed size of a snapshot taken for each of these apps after we used the application for a short period, for example, playing RAPT or scribbling on the SketchPad for a minute. These actions taken by a user do have some effect on the snapshot size, but for the most part, the size does not vary greatly over time (this is shown further in Section 5.4). This is true unless a large amount of new data is actually generated by the user, e.g. finishing a detailed painting in Sketchpad. However, it that case we expect the user would want that data saved anyway, so it probably wouldn't be considered as an overhead imposed by Imagen. For three of the four apps, the snapshot size was about 2MB or less. This amount of data is likely to be saved and transferred by any user without any problem. We believe even the case of ColorPiano at 8MB is still modest. It is certainly possible other apps we have not tested vary greatly from these numbers although these apps do cover a range of different web app use cases.

5.4 Snapshot Size vs. Mugshot Log Size

In theory there are two ways to recreate the state of a program: state-capture or event-playback. Imagen works through state-capture: at a given instant in time the complete state is recorded to stable storage. On the other hand, Mugshot [27] is an existing approach that provides event-playback for Web applications: each event that occurs during program execution is recorded in a log and the log can be used later to playback the application to some state. State-capture has the advantage that recording size is limited by the size of the application state. Event-playback has the advantage that a user can return to any point in the history of execution.

Here we address our fourth research question and demonstrate why state-capture is a more practical solution for supporting browser session migration. The evaluation is not intended to show that Imagen is better than Mugshot, because Mugshot was not even designed as a tool for session migration. Mugshot uses event-playback because that approach is useful for the purpose of debugging. Regardless of this evaluation, Mugshot would continue to be a better tool for its intended purpose of debugging. However, since our focus is on migration, we show why Imagen is practical in this regard.

The problem with event-playback, which we demonstrate here, is that the event log grows without bound as execution continues. For the purpose of session migration, this would create a limit on the amount of time a user can use an app before migration became intractable. The only way to avoid this problem is to create a checkpoint which captures the state at some point, relieving the need for the event-log up to that checkpoint. This is precisely what Imagen does.

In a previous paper on Mugshot, measurements were provided which show the growth rate of the size of the event logs for some apps. Here we compare the size of a Imagen snapshot for the same apps versus the size of those logs as they grow over time.

The first comparison is provided for a PacMan clone [10]. The log growth rate during gameplay was given by Mugshot as 75kB per minute. This means that after four minutes the log size for Mugshot would already be 300kB whereas in our measurements the Imagen the snapshot size continues to be roughly 25kB regardless of how long the game is played. While 300kB is not large, this trend places an unnecessary time limit on the user. Also, this size is modest because this application only relies on keyboard events which are usually less frequent than mouse events (mouse events must continuously track screen position). The next comparison shows how the case is worse for these commonly used event types.

This second comparison is provided for a paint program called Canvas Painter [33]. The log growth rate was given by Mugshot as 795kB per minute. This is shown in Figure 8 interpolated over the period of four minutes. For Imagen, we took the snapshot size at each one minute interval, over a period of four minutes. The activity provided as input is simply random pencil drawing on the canvas. Now we see a large difference as the Mugshot log grows to 3MB in

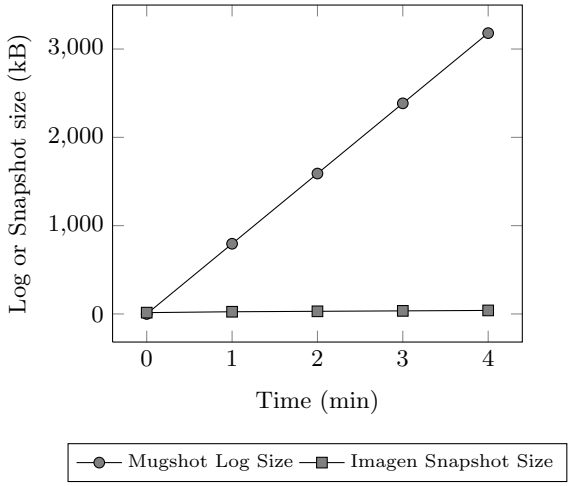

Figure 8: Imagen snapshot vs. Mugshot Log Size for Canvas Painter.

only four minutes, whereas for IMAGEN the snapshot size grows slowly. After only thirty minutes, the event logging approach would most likely be impractical for session migration. While difficult to see in the graph, the IMAGEN snapshot grows from 23kB to 26kB over the four minute interval.

While it is possible for an event log to be more efficient than a state snapshot at the early part of program execution, event logs will always grow without bound. A combination of checkpointing and event logging could possibly bring some benefits for session migration, however, this was not needed for any of the applications we looked at and IMAGEN's approach never resulted in a snapshot larger than 9MB. This evidence suggests that for the specific case of session migration, state capture is a more practical approach.

6. RELATED WORK

State synchronization. State synchronization can be seen as a special form of state migration. For instance, Amazon Kindle's WhisperSync and Chrome browser's sync support synchronizing bookmarks across different devices. Online tools for collaborative editing or browsing synchronize the DOM's tree structure [6, 16, 19, 26]. Sync Kit [4] and Replets [37] replicate server data of apps to the browser but do not handle other runtime state such as closures or events. WedPod [31] offers cross-device browsing by implementing a special purpose Linux-based browser that persists a session. IMAGEN runs in JavaScript on existing browsers across multiple devices.

Deterministic Replay. Deterministic replay is primarily used by event-capture/replay testing and debugging tools [2, 27]. These tools allow advanced developers to revisit any recorded program state to track down bugs. However, deterministic replay also requires constantly logging the application's dispatched events. Our evaluation indicates that these logs are not likely to be practical for use by end users in session migration.

Object Serialization and Persistence. As described in the introduction, using a persistence library [4, 7, 30], the programmer still has to manually register and manage individual objects. Moreover, persistence is a cross-cutting concern, so scattered changes are need over the application's code, making it less maintainable.

The automated and generic nature of our instrumentation makes our approach transparent and helps developers keep their original source code intact. While our previous work [7] helped to automate database tasks related to persistence of JavaScript objects, it did not handle function closures, event-handlers, media objects, or the other challenges we addressed in this paper related to the migration of running browser sessions. Similarly, previous work on Web app migration [3] did not address these issues.

Server Side Process Migration. Server side process migration [28, 14, 25] transfers a system level process between two machines and is used for administration tasks such as load balancing and fault-resilience. Process migration happens in the kernel so implementation is quite complex. Such systems work at a lower-level of abstraction handling issues such as memory page allocation and thread scheduling. Our work focuses on a new domain where the Web browser becomes the "operating system". Traditional process migration which suffered from slow adoption due to difficulty adapting to different platforms. Since technologies such as HTML5 are being standardized, interoperability for browser session migration is easier to achieve.

7. FUTURE WORK AND CONCLUSION

We have presented a generic solution to session migration, which works in the JavaScript layer and also targets some HTML5 APIs. However there are still other APIs which are not covered by our current implementation, such as WebWorkers and GeoLocation. WebWorkers provides support for background computational tasks but since each worker has an isolated memory and cannot respond to UI events, our assumptions made in this paper that depend on a single-threaded model still apply. While additional effort will be required to enable support, the fact that such APIs are being standardized should help making migration support feasible. We are also looking into source code that are being generated by eval().

While we have not focused on debugging in this paper, it may be possible to use IMAGEN so that when an end user requires urgent assistance, she can instantly duplicate and share a session snapshot with developers who could inspect the state in a web developer tool such as FireBug. We plan to investigate such debugging support also in our future work. We hope to combine IMAGEN with the Mugshot approach to bring checkpointing functionality to an event-logging debugger.

In this research, we tackled technical challenges that arise in the context of web technologies where state is created and managed at the user's browser. This occurs frequently in apps that make use of Ajax and HTML5. In order to make such state data ubiquitously available to users, we investigated an approach based on session migration, a conceptual descendant of previous research on process migration. Our solution using JavaScript transformation has the advantage that it does not require additional coding. From the evaluation the overhead seems reasonable and quite low on several apps and browsers. We also showed that an approach to session migration based on event-playback is not likely to be practical compared to our state-capture approach.

8. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] S. Andrica and G. Candea. WaRR: A Tool for High-Fidelity Web Application Record and Replay. In *Proc. of Dependable Systems and Networks*, 2011.

[3] F. Bellucci, G. Ghiani, F. Paternò, and C. Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proc. of the Symposium on Engineering interactive computing systems*, 2011.

[4] E. Benson, A. Marcus, D. R. Karger, and S. Madden. Sync kit: a persistent client-side database caching toolkit for data intensive websites. In *Proc. of WWW*, 2010.

[5] R. Berjon, T. Leithead, E. D. Navara, E. O'Connor, and S. Pfeiffer. W3C HTML5, 2012. http://dev.w3.org/html5/spec/.

[6] I. Bicking. Browser mirror. `https://browsermirror.ianbicking.org`.

[7] B. Cannon and E. Wohlstadter. Automated object persistence for JavaScript. In *Proc. of WWW*, pages 191–200. ACM, 2010.

[8] HTML5 Canvas. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.htmlp`.

[9] C. Cat. Convergence. `http://currantcat.com/convergence`.

[10] K. Cieslak. Pacman! `http://www.digitalinsane.com/api/yahoo/pacman/`.

[11] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[12] M. Deal. Colorpiano. `http://mudcu.be/piano/`.

[13] M. Deal. Sketchpad. `http://mudcu.be/sketchpad/`.

[14] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21:757–785, 1991.

[15] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[16] N. Fraser. Differential synchronization. In *Proc. of the Symposium on Document Engineering*, pages 13–20, 2009.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] Genoverse.org. Genoverse - html5 genome browser. `http://www.genoverse.org/`.

[19] M. Heinrich, F. Lehmann, T. Springer, and M. Gaedke. Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proc. of WWW*, pages 1057–1066. ACM, 2012.

[20] A. L. Hosking and J. Chen. Mostly-copying reachability-based orthogonal persistence. In *Proc. of Object-oriented programming, systems, languages, and applications*, pages 382–398, 1999.

[21] JitBit. Macro Recorder Lite. http://www.jitbit.com/macro-recorder-lite/.

[22] Introducing JSON. `http://www.json.org/`.

[23] JsonML. JSON Markup Language. http://www.jsonml.org.

[24] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[25] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *International Conference on Distributed Systems*, 1988.

[26] D. Lowet and D. Goergen. Co-browsing dynamic web pages. In *Proc. of WWW*, pages 941–950, 2009.

[27] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for Javascript applications. In *Proc. of Networked Systems Design and Implementation*, pages 159–174, 2010.

[28] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 2000.

[29] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[30] PersistenceJS. `http://persistencejs.org`.

[31] S. Potter and J. Nieh. Webpod: Persistent Web browsing sessions with pocketable storage devices. In *Proc. of WWW*, pages 603–612, 2005.

[32] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proc. of Aspect-oriented software development*, pages 120–129, 2003.

[33] R. Robayna. Canvas painter. `http://caimansys.com/painter/`.

[34] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of Object-oriented programming, systems, languages, and applications*, pages 174–190, 2002.

[35] E. Wallace, J. Ardini, K. Gishen, and P. Kernfeld. Robots are people too. `http://raptjs.com/`.

[36] The Open Web Application Security Project. WebScarab. `https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`.

[37] D. Zhou, N. Islam, and A. Ismael. Flexible on-device service object replication with replets. In *Proc. of WWW*, pages 131–142, 2004.