공학박사 학위논문

Exploiting Snapshot for Web Applications

웹 어플리케이션을 위한 스냅샷 활용

2016 년 2 월

서울대학교 대학원

전기 컴퓨터 공학부

오 진 석

Exploiting Snapshot for Web Applications

웹 어플리케이션을 위한 스냅샷 활용

지도 교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함
2015 년 12 월

서울대학교 대학원
전기 컴퓨터 공학부
오 진 석

오진석의 공학박사 학위논문을 인준함
2016 년 1 월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

위 원 윤 성 로 (인)

위 원 김 수 현 (인)

Abstract

Exploiting Snapshot for Web Applications

JinSeok Oh

School of Electrical Engineering and Computer Science

The Graduate School

Seoul National University

Web applications (apps) are programmed using HTML5, CSS, and JavaScript, and are distributed in the source code format. Web apps can be executed on any devices where a web browser is installed, allowing one-source, multi-platform environment. We can exploit this advantage of platform independence for a new user experience called *app migration*, which allows migrating an app in the middle of execution seamlessly between smart devices. We proposes such a migration framework for web apps where we can save the current state of a running app and resume its execution on a different device by restoring the saved state. We save the web app's state in the form of a *snapshot*, which is actually another web app whose execution can restore the saved state. In the snapshot, the state of the JavaScript variables and DOM trees are saved using the JSON format. We solved some of the saving/restoring problems related to event handlers and closures by accessing the browser and the JavaScript engine internals. Our framework does not require

instrumenting an app or changing its source code, but works for the original app. We implemented the framework on the Chrome browser with the V8 JavaScript engine and successfully migrated non-trivial sample apps with reasonable saving and restoring overhead. We also discuss other usage of the snapshot for optimizations and user experiences for the web platform.

Another issues of web app is its performance. Web apps are involved with a performance issue due to *JavaScript*, because its dynamic typing, function object, and prototype are difficult to execute efficiently, so even just-in-time compilers do not help much.

We propose a new approach to accelerate a web app, especially its *loading time*. Generally, running an app is composed of app loading to initialize the app, followed by event-driven computation. If the same job needs to be done to load an app, especially the execution of the same JavaScript code, it will be better to save the JavaScript execution state in advance and to start the app from the saved state. In fact, app loading is often involved with the initialization of the web framework such as jQuery [1], Enyo [2] , or Ext JS [3] where many JavaScript objects are created. Also, app-specific objects are created during app loading. If we save the initialized state of these objects in the form of a *snapshot* and start app loading by restoring the objects from the snapshot, we would accelerate app loading.

Snapshot which saves JavaScript execution state has limitation because it cannot save DOM state. By using DOM Log-Replay which saves DOM state, we can save whole app loading state. Thus, we can remove all JavaScript execution during web app loading and directly reach the state which loading is completed

by DOM snapshot, resulting in acceleration of web application loading time. DOM Log-Replay saves every DOM-related action occurs during loading time as a log. When restoring snapshot, log is replayed to restore DOM tree and various DOM elements. We actually implemented the idea for the web applications based on two web framework. This can reduce the whole app loading time by at least 60%, which could be noticed tangibly.

Because snapshot saves JavaScript state as file, there is size overhead. This can be an issue in device with small memory such as IoT. Thus, size of snapshot needs to be reduced. Firstly, we compress snapshot and hide decompression overhead by implementing pre-decompression which decompresses snapshot in another thread before snapshot deserialization. Secondly, we implemented framework snapshot sharing which shares snapshot of JavaScript framework between apps developed with the same framework. With these techniques, performance is degraded for about 1.7%, but memory overhead is dramatically reduced from 12x to 2.59x.

Keywords : web app, JavaScript, snapshot, web framework, loading-time, app migration
Student Number : 2009-20839

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1. Web Applications

For the last few years, many smart devices such as phones, tablets, TVs, and car infotainments have been actively used, especially by running diverse applications (apps) on top of those devices. Currently, Android apps or iOS apps constitute the main stream of the app platform, but a new platform of apps called *web apps* would soon join the mainstream with Tizen [4], webOS [5], and Firefox OS [6]. These *web platforms* have the advantage of portability and productivity compared to existing platforms. That is, web apps can be executed on any devices where a web browser is installed, supporting one-source, multi-platform environment. Moreover, programming based on the existing web technology allows a faster app development.

Web apps are programmed using HTML, CSS, and JavaScript. HTML expresses web components, CSS controls visual effects, and JavaScript performs computation and manipulation for the apps. The browser parses the HTML document and builds a document object model (DOM) tree, which is then displayed on the screen based on CSS by the rendering engine. The JavaScript engine executes the JavaScript code in the web app, mostly for handling the events and manipulating the DOM tree based on the events, which is then rendered for an updated display. Recently, HTML5 has been introduced for implementing multimedia components without plugins such as Adobe Flash [7]. Also, HTML5 provides APIs to control the hardware components of smart devices such as battery or camera

[8], allowing the programmer to develop a "complete" application within the boundary of web technology.

Despite the advantage in portability and productivity, web apps are involved with some performance issues, mostly due to JavaScript. First, JavaScript should be parsed and interpreted at runtime. Also, unlike Objective-C (iOS) or Java (Android), JavaScript supports dynamic typing, prototypes instead of classes, and first-class function objects created at runtime, all of which make execution extremely inefficient. To improve the performance loss on JavaScript interpretation, just-in-time compilation (JITC) is used to compile JavaScript code to machine code at runtime. JITC often uses runtime profiling to resolve types, which is effective when the operand types do not change at runtime, or employs hidden classes to allow offset-based property accesses, which is effective when the object structure is not changed. Unfortunately, JITC based on these techniques works effectively on benchmarks, but not on web apps or web pages. Real apps are more dynamic than the benchmarks, so types or prototypes can change [9]. Also, app functions are called fewer, and app loops iterate fewer than those of benchmarks, so JITC suffers more from the compilation overhead [10]. In fact, when we turn on JITC for web apps, we rarely see any performance improvement.

1.2. Snapshot for Web Applications

Snapshot is a saved app execution stated at specific time. Originally snapshot is used to save history of deal with fault tolerance in database. Snapshot for web app is saving specific execution state of web app, which can be used for various purposes. We used snapshot for two purposes. First, we implemented app migration which shares

execution state between devices [11]. App migration is to send app state from device user is using to other device in the middle of runtime so that user can continuously execute app in the other device. Second, we implemented loading-time acceleration by saving initialized state of app, which is the state app loading is completed, and loading app with snapshot when app loading is needed [12].

Some issues should be considered in order to use web application snapshot for various purposes. First, we have to consider what to save. Diverse states occur while web app is executed. For example, there can be DOM tree or JavaScript state. Some of these state cannot be saved, and others do not need to be saved for state restoration of web app. So we have to choose which state to save for snapshot's purpose. Second is which format to save snapshot. In case of app migration, snapshot can be used in various devices and browsers, cause it should cover many devices. However, in loading-time acceleration, low-level saving format is needed to restore state quickly.

Third is when to save. For app migration snapshot should be saved while user is using the app. There can be some point which saving is impossible or very inefficient. For loading-time acceleration, snapshot should be saved in the point which loading is done as much as possible in order to fully optimize. So we should concern about saving point.

1.3. Organization of the thesis

This paper deals with how to use web application snapshot for web app. The rest of the paper is organized as follows. Section 2 explains how web app is loaded and executed, showing which states occur in web app and what should be considered when saving these states. In

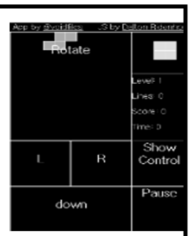
Section 3, we proposed app migration using web application snapshot which can provide novel user experience. Section 4 describes first study of JavaScript snapshot to improve web app loading time using web application snapshot. Section 5 describes enhanced snapshot study which can overcome limitation of JavaScript snapshot: DOM state saving using DOM Log-replay, Pre-decompressing and Framework Snapshot sharing which minimizes memory overhead of snapshot. Related works are mentioned in Section 6 and Section 7 concludes.

Chapter 2. How Web Apps Work

2.1. Web App Loading

We first overview how web apps work. As a web page, a web app is run by executing an HTML file, embedded with JavaScript code fragments and the CSS. The web browser allocates a global object called window, which manages all the elements displayed for the HTML page. The window object has a property (which is similar to a field of an object) called a document, which is the root of the DOM tree, a format standardized for the document object. Each DOM tree node corresponds to an element included in the HTML page such as text, image, and video component. The browser reads the HTML file and parses all of its components to build a DOM tree. Each component of the HTML page is separated by the tag such as <head> or <title>, so the HTML parser can identify each component and its hierarchy, which is added to the tree. The DOM tree is displayed with a visual effect of the CSS. When a script tag is encountered during the HTML parsing, the corresponding JavaScript global code is executed. When the last HTML tag is parsed and the onload event is fired by the browser, the loading process of the web app completes. Now the web app proceeds in an event-driven manner, such that a JavaScript function registered as an event handler is invoked when an event occurs such as the mouse click, the keyboard input, or the timer event. The event handler often changes the DOM tree for an updated display such that a DOM node belonging to a DOM tree can be changed or a new one is added to the tree via JavaScript execution using the DOM APIs.

HTML(index.html)




```
1  <html manifest="tetris.manifest">
2  <head> …  <link href="tetris.css" … > </head>
3  <body>
4  <div id="tetris">
5  ...
6  <div class="game_area"> <div id="canvas">
7  <div class = “square typ3” style=“left: 100px; …”></div>
8  <div class = “square typ3” style=“left:  60px; …”></div>
9  <div class = “square typ3” style=“left:  80px; …”></div>
10 <div class = “square typ3” style=“left: 100px; …”></div>
11 </div></div>
12 <div id="info">…<p id="lines">Score: … </div>
13 </div>
14 <div id="controls" class="show_controls">
15 <div class=…> <div class=…><span>L</span></div></div>
16 ...
17 </div>
18 <script type="text/javascript" src="tetris.js"></script>
19 </body>
20 </html>
```

Figure 1. Example HTML code of the *tetris* web app

For example, consider a simplified Tetris app shown in Figure 1 [13]. Its HTML file (index.html) is composed of HTML tags to describe the title, the game score, and the space information for the rectangle blocks, etc. (line 4−13). The CSS file is also included (line 2) to show the style information for the block color/shape and the layout of the game screen. A JavaScript file in Figure 2(tetris.js) is invoked (line 18) for controlling the game actions. It declares a global variable tetris and creates an object (line 1). Many objects including function objects such as init() or play() are created and saved as the properties of the tetris object (line 2−30). Other global variables are also declared and initialized (line 31). In this paper, we use global variables and global objects interchangeably, and they mean those JavaScript objects accessible from the global variables of the window

```
 1  var tetris = new Object();
 2  tetris.canvas = null, tetris.controls = null, …
 3  tetris.init = function() {
 4    this.canvas=document.getElementById("canvas");
 5    ...
 6    this.bindControlEvents(); this.play();
 7  },
 8  tetris.bindControlEvents = function() {
 9    var cb = function(e) { tetris.handleControl(e); }
10    this.controls = document.getElementById("controls");
11    this.controls.onclick = cb;
12  },
13  tetris.gameOver = function() {
14    ...
15    this.canvas.innerHTML = "<h1>GAME OVER</h1>";
16  },
17  tetris.play = function() { //gameLoop
18    var me = this;
19    ...
20    var gameLoop = function() {
21      me.move('D');
22      me.pTimer=setTimeout(gameLoop,me.speed);
23      ...
24    };
25    this.pTimer=setTimeout(gameLoop,me.speed);
26  },
27  tetris.incScore = function(amount) {
28    score = score + amount;
29    this.setInfo('score');
30  },
31  var score = 0, level = 1, lines = 0;
32  ...
33  tetris.init();
```

Figure 2. Example JavaScript code of the *tetris* web app

object, which differ from local variables (objects) in a function. Finally, the function tetris.init() is invoked (line 33).

During the execution of tetris.init(), a DOM element created in the HTML file (line 14 of tetris.html) with an ID of "controls" is accessed using a DOM API, getElementById(), so that a reference to the element is saved in tetris.controls (line 10 of tetris.js). We call this variable a DOM reference variable. Then, an onclick event is

registered for this DOM reference variable (line 11) so that when there is a mouse click on the control panel, the registered event handler cb() is invoked to move left/right (see Section 3.4.4). When tetris.play() is executed, a timer event is also registered (line 22, 25) using a closure variable (see Section 3.4.2), so that the current block moves down repetitively with some time interval. These event handlers will change the location of the current block using the move() function, so the DOM tree will show an updated shape when it is rendered. The function tetris.gameOver() will change the DOM tree by adding an innerHTML component when invoked to show the "Game Over" string. Some functions such as tetris.incScore() will update a global variable when invoked.

Above description indicates that the execution state of a web app includes the DOM tree state and the JavaScript execution state. Both are the properties of the window object. Another property is the browser information. So, the window object has a property structure in Figure 3(key0, key1, … are global variable names).

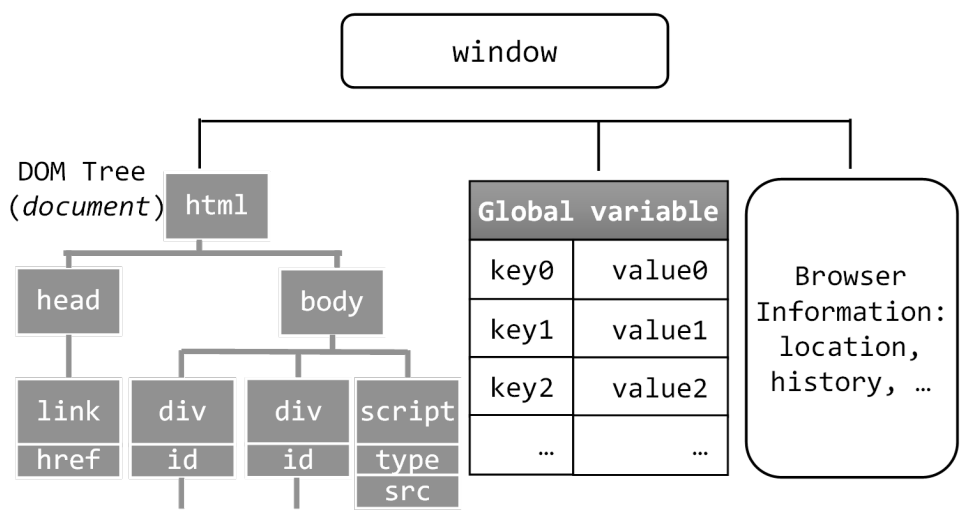


Figure 3. window object structure

2.2. JavaScript Execution State

As to the JavaScript state, we first understand how the JavaScript works with its runtime data structures. The JavaScript engine is invoked when the first script tag is met during the HTML parsing. A heap area is assigned for it, and initialization is performed to make all built-in objects (which are also global variables) such as math objects in the heap. A runtime data structure called an *execution context* (EC), which roughly corresponds to the activation record in C, is generated in the heap when the JavaScript code is executed. A global execution context (GEC) is generated first, which contains all properties of the *window* object including the DOM tree, JavaScript global variables including those made by the programmer and those built-in variables, and the browser information. When a JavaScript function is invoked, its EC is also generated which contains its local variables and parameters. These ECs form an EC stack with GEC at the bottom, which correspond to the call stack of C. Independently, each function has a *scope chain* employed to find a variable name during execution, which is a linked list of ECs with GEC at the end. JavaScript requires the scope chain since it allows inner functions, so an inner function can use variables defined in outer functions; the scope chain is searched linearly to find a name. In real implementation, however, the scope chain is used only for the closure function to retrieve the closure variables (see Section 3.4.2) because other outer variables can be resolved when the JavaScript code is parsed. The JavaScript global code or function is parsed to the intermediate representation (IR), which is interpreted or JIT-compiled to machine code for execution. Figure 4 illustrates the EC stack and the scope

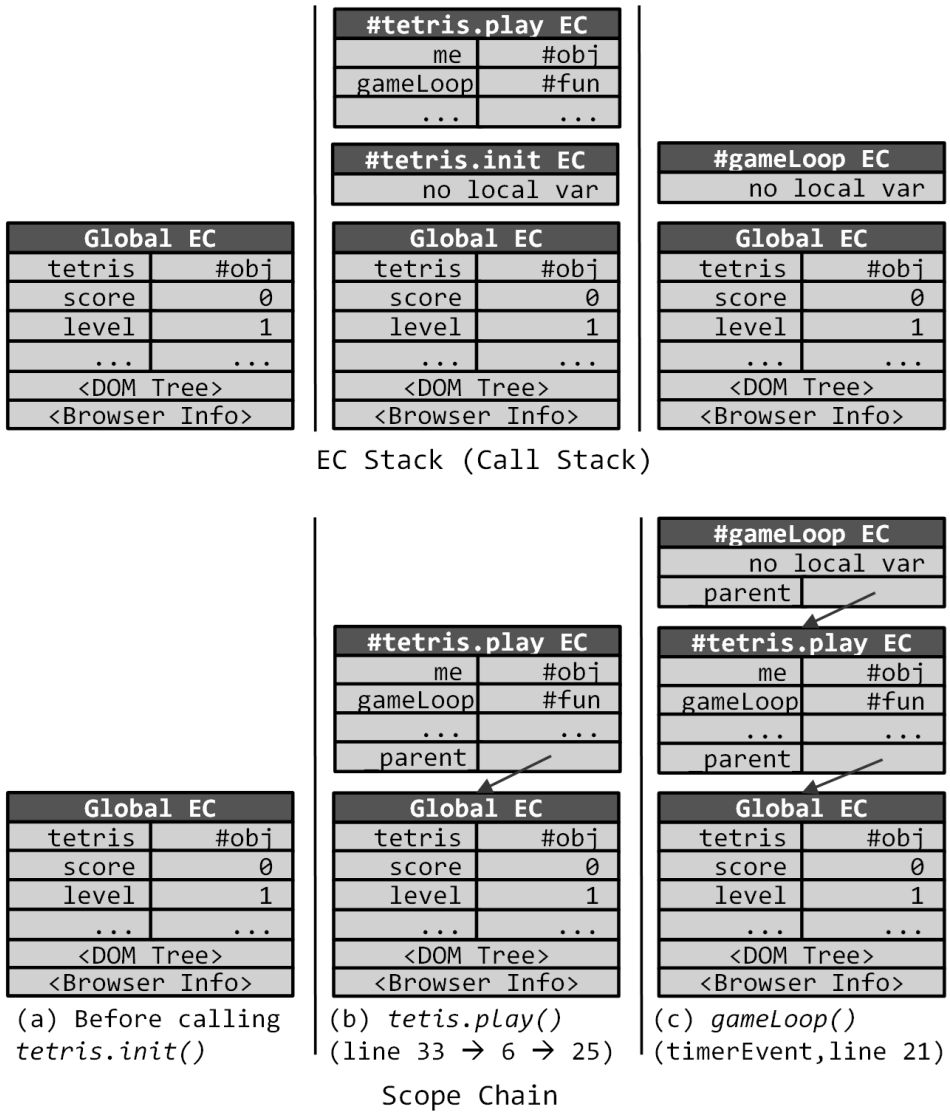


Figure 4. EC stack and scope chain for Figure 2

chain, when *tetris.play()* and *gameLoop()* are in execution in Figure 2.

Chapter 3. Migration of Web Applications

3.1. App Migration

Text-based source code distribution of the web apps may lead to a new user experience (UX) beyond the advantage of portability. That is, it would be interesting to move an executing app on a device to a different device and continue its execution seamlessly, which we call *app migration*. For example, we can migrate a game app being played on a smart phone to a smart TV and continue to play with it. Also, we can move a secretary app worked on a smart TV to a tablet, then to the in-vehicle infotainment (IVI) of a smart car for continuous secretary services on the move.

This app migration would also be possible if the app developer programs an app manually considering the migration between devices, or if a proxy server is used to add some instrumentation code to the app source code [14] [15]. However, it would be more useful if app migration works transparently for *any* app, supported entirely by the web browser, as in mine. my proposed app migration is a more elaborate and convenient extension of the existing services such as the *chrome tab sync* to allow all the tabs of a browser on a desk-top to be browsed on a mobile browser [16], or the *dropbox* cloud service to access pictures and videos between devices [17]. App migration might also be possible for Android or IOS, but web apps allow easier extraction of properties from objects, and the text-based portability makes migration simpler, as explained shortly.

We proposes an app migration framework for web apps where we can capture a running app by saving its current state, and resume the app by restoring the saved state on a different device. We save the web app's state in the form of a *snapshot*, which is actually

another web app composed of HTML/CSS/JavaScript, and if we execute the snapshot, it will restore the saved state automatically. That is, the snapshot has a text form where many of the web app states are saved in a platform-independent JSON format [18]. This is in sharp contrast to existing job migration used in the system VM where an app state on a desktop is migrated to a mobile device using VMware image [19]. Also, it is different from hibernation, where the whole process states are stored [20]. These techniques save the whole system image or whole process images, requiring a much higher memory overhead and restoration complexity. Our app migration saves the state of a single app, only the essential parts of the execution state after finding an appropriate time to save. Platform-independent text-form of the snapshot allows easier and more efficient migration between diverse devices with different CPUs, OS, browsers, and JavaScript engines, with a simple click of a button on the browser.

3.2. App Migration Framework and Scenario

The current implementation of my app migration framework exploits the *extension* feature of a browser [21], a small program for enhancing the functionality of a browser. It can be programmed using HTML, CSS, and JavaScript, and be added as a button in the browser.

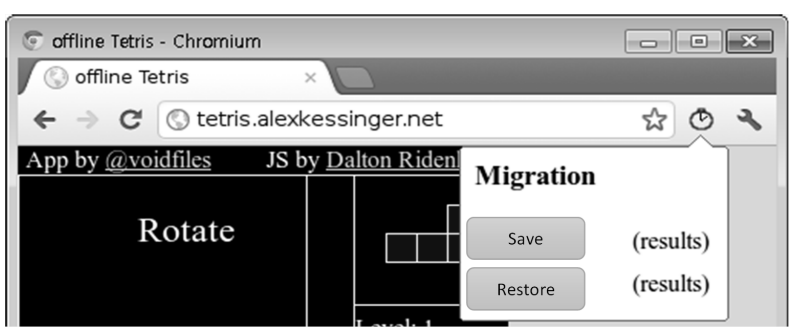


Figure 5. Browser extension of the Chrome browser

For Google's Chrome browser, for example, a small button can be added at the end of the address bar as shown in Figure 5, which can be clicked for the execution of the extension. Other browsers require a different implementation of the extension, yet it is similarly programmed. We do not need a separate proxy server for app instrumentation [14] [15], but just click the button to save the app state. We might need a more intuitive user interface as the swiping-based one used in Apple's Airplay [22] for saving, transferring, and restoring in real time, yet it is beyond the scope of this paper.

App migration framework works as follows. A user runs a web app by pressing its icon installed in the device (platform-based app) or by entering an URL in the web browser (URL-based app). During execution, if the user wants to save the current execution state for app migration, he simply clicks the save button to execute the extension for the saving. The current state will be saved in the snapshot file, which will then be transferred to a different device.

The app restoration works similarly. The user runs an app by pressing its icon or entering its URL, but this time the user clicks the restore button of the extension to resume the execution of the migrated app. The snapshot file in the local storage is read and executed, which will resume the app execution seamlessly.

There can be variations on the scenario. As mentioned previously, the snapshot itself is a web app composed of HTML/CSS/JavaScript, so the user can directly run it on the browser without the extension. It might also be possible to save the files on a remote cloud server when we save the app state using the extension, which are then downloaded and executed when we want to resume its execution on a different device.

3.3. Saving and Restoring the App State

A naïve way of saving the JavaScript execution state would be simply saving all the JavaScript global objects in the GEC, all other objects, all ECs in the call chain, the scope chains of closure functions, IRs, and even the JIT-compiled machine code. However, the amount of data to save and the time overhead would be substantial. More seriously, the restoration process will be complicated since all of these data structures should be relocated in the target machine, and restarting the JavaScript code in the EC stack would be tricky, especially if it is JIT-compiled. The process might require same JavaScript engine, same browser, or even same target machine. Maybe the app should also be installed in the target device in advance. Therefore, this approach is not convenient for app migration.

We take a different, simpler approach. Instead of saving the data structures as they are in the heap, we save them in a form of JavaScript code such that when it is executed, equivalent data structures will be restored automatically. This means that our snapshot is actually a JavaScript file. For example, if a global variable *x* has a value *0* when we save, we generate a JavaScript statement *var x=0;* in the snapshot file, so if it is executed at the target device, it will restore the GEC with *x* set to *0*. All the global variables can be restored in this way. However, the function execution state cannot be easily saved to JavaScript code. Fortunately, a JavaScript function as an event handler tends to be executed briefly to provide a fast user response time. So it is better to wait for an event handler to finish before we save, rather than saving in the middle of an event handler execution. That is, we take a snapshot only when no JavaScript code is in execution, in-between the execution of

JavaScript event handlers. This would obviate the saving of the function execution state, so we need to save only the JavaScript global variables (and the DOM tree), shared among the event handlers.

For the *Tetris* app example, the game execution after loading is composed of the execution of a timer event handler that moves the current block downward, and the execution of a DOM event handler that manipulate the current block based on the mouse click. Between the execution of these event handlers, the DOM state and the JavaScript state do not change, thus being appropriate and safe for saving the app state.

The saving job itself will be done by a JavaScript function, *state_save()*, which we added for the browser extension. It will be executed when we click the save button. This function can access the *window* object of the app, so it can access all the global variables and the DOM tree. So, *state_save()* will collect all JavaScript global variables. This is possible since the extension mechanism allows the added JavaScript function to share the same browser session with the app when it is declared in *manifest.json*.

Before We describe how state_save() works, we first discuss the format used to save the JavaScript objects pointed by the JavaScript global variables. We use the JavaScript Object Notation (JSON), which is a string format used to send data between web applications using AJAX (Asynchronous JavaScript and XML) or web services, and is useful to represent the objects in many programming languages [23] [24] [25]. Figure 6 (a) shows an example of JavaScript code and Figure 6 (b) shows the corresponding JSON-based JavaScript code. So, if we take a snapshot at the end of execution for the JavaScript code in Figure 6 (a) and generate the

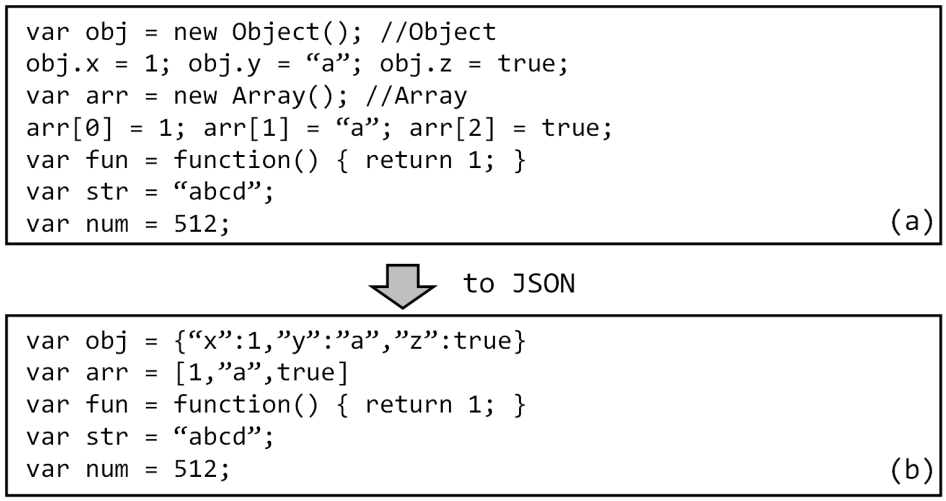


Figure 6. JSON format example

snapshot file as in Figure 6 (b), the execution of the snapshot file will generate the exact same JavaScript state. We can actually apply *JSON.parse()* to the JSON string (e.g., *var arr = JSON.parse([1,"a",true])*) as will be explained shortly, yet the result is the same. JSON is simple and supported by all browsers, thus useful for app migration.

The process of converting a JavaScript object to a JSON format string is called *serialization* (the opposite is *deserialization*) [26]. JavaScript provides an API for serialization (*JSON.stringify()*) and for deserialization (*JSON.parse()*). So the state saving and restoring can be implemented by invoking these APIs, as explained below.

Figure 7 (a) shows a sketch of the JavaScript function, *state_save()*, invoked when clicking the save button in the browser. The function accesses each global variable property of the *window* object, serializes it using *JSON.stringify()*, and saves its name and JSON string to an array. Finally, it creates an output file using the array. The file includes not only the JSON strings, but the JavaScript code where *JSON.parse()* is invoked for the JSON strings, as shown in Figure 7 (b). So, it is actually a JavaScript file (*snapshot.js*), and

state_save()

```
function state_save(window) {
 //argument is window object
 var result = new Array();
 //for store JSON result
 for (var key in window) {
  //For each JavaScript global variable
  //name key of the window object
  var json = JSON.stringify(window[key]);
  //window property (global variable) to JSON
  result.push([key,json]);
 }
 makeOutput(result);
 //make output file(snapshot.js & snapshot.html)
 //based on result
}                                                    (a)
```

snapshot.js

```
var {key1} = JSON.parse(<JSON string>);
//load value stored based on <JSON string>
var {key2} = JSON.parse(<JSON string>);
…                                                    (b)
```

snapshot.html

```
<html>
<head> …
<script type="text/javascript"
          src="snapshot.js"> </script> </head>
<body> </body>
</html>                                              (c)
```

Figure 7. Simplified pseudo code for *state_save()* and the snapshot JavaScript and HTML files

if it is invoked, the saved JavaScript state can be restored as each JSON string is deserialized to a JavaScript object.

Another part of a web app state is the DOM state. Unlike the JavaScript global objects, *JSON.stringify()* does not work for the DOM objects since the DOM tree is a native property located in the web browser. Instead, we use the *JsonML* library to convert between the DOM tree and the JSON string [27]. So, we add the JSON string for the DOM tree obtained using the *JsonML* library, and a *JsonML* library call with the string as an argument, to *snapshot.js* (see Section

3.4.3). When this JavaScript file is executed, the DOM tree as well as the JavaScript global objects will be restored.

An HTML file is also generated (*snapshot.html*) as in Figure 7 (c), which will do the invocation of the JavaScript file. These two snapshot files are, in actuality, a legitimate full-fledged web app runnable on any browser. That is, our approach saves the execution state of a web app A by creating a new web app A' which evolved from the original app A, but captures the execution result so far. All of the JavaScript global objects (including the function objects) and the DOM tree will be included in the snapshot JavaScript file so that they are restored when the file is executed.

Previous work also used the JSON format using the *JSON.stringify()* and generate JavaScript code for restoration [14] [15]. However, they modify the original JavaScript source code by adding instrumentation statements, and run the instrumented app to save the execution state, instead of the original app. The additional code retrieves the internal information needed to serialize the app state correctly. In our scheme, we run the original app and retrieve the internal information by accessing the browser and the JavaScript engine data structures. We think this is more practical for a web platform, which often requires a browser enhanced with additional features, so adding these new access APIs to the browser will be more straightforward than instrumenting every app. Moreover, we can pursue the next step of app migration based on a binary format for faster restoration, as will be discussed later. Our approach is also more advantageous in security (see Section 3.4.2), and allows generalizing the snapshot for other optimization and user experience purposes (see Section 3.7).

There are a few issues in saving and restoring the app state, which will be discussed below.

3.4. Issues for Saving and Restoring the App State

The previous section overviews our approach to saving and restoring the app state. There are a few technical issues to be solved, which will be presented in this section.

3.4.1 Object Reference Alias Problem

The existing *JSON.stringify()* cannot keep the original object reference information when it serializes an object to a JSON string,which can cause an alias problem when we deserialize, as often complained by the web programmers [28] [29] [30]. For example, the two variables, *obj1* and *obj2* in Figure 8 (a) point to the same object, but when they are serialized by *JSON.stringify()* as in Figure 8 (b), they are represented by two separate JSON strings. The problem is that when they are deserialized, they will be restored to

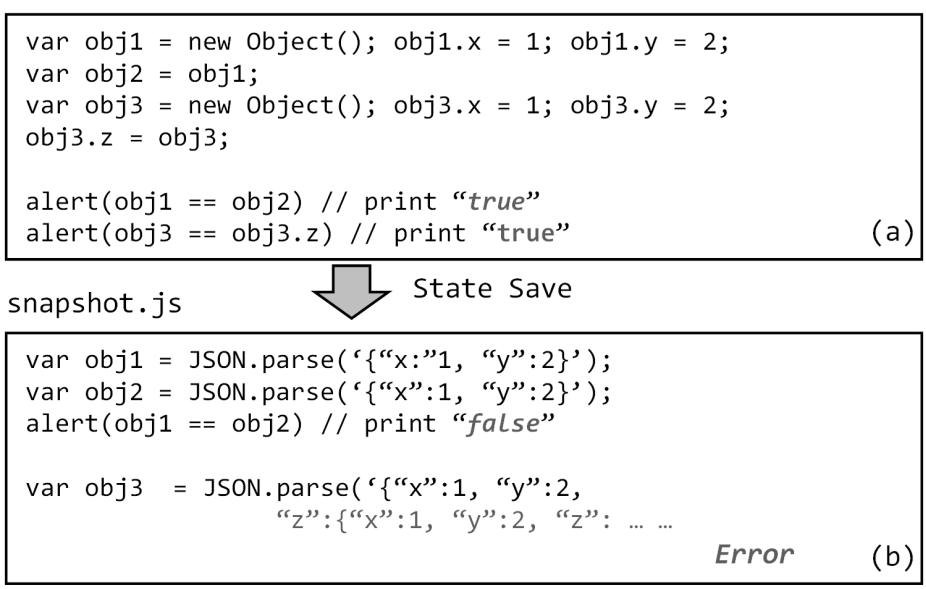


Figure 8. Reference alias and circular reference problems

two separate objects, which can make the program behave differently as shown in Figure 8. A similar problem can occur for a circular object structure as *obj3* in Figure 8 (a) where a property of an object points to itself; serialization for the object will fail with an exception as shown in Figure 8 (b) since it cannot handle recursive serialization.

These problems occur since the original reference information is lost during serialization. So, we must keep track of the reference information when we save, to avoid duplicate serialization. We use an

```
JSON.stringify = function(arg) {
 if(arg.isObject()) {
  var index = sobj_ref.indexOf(arg);
  //check if arg is already in sobj_ref
  if(index == -1) { //arg is not in sobj_ref
   index = sobj_ref.push(arg); //register arg to sobj_ref
   var json = toJSON(arg); //serialize arg to JSON string
                          //(recursively for its properties)
   sobj_ref_value[index] = json;
  }
  return "dobj_ref[" +index + "]";
  //return string with dobj_ref index
 }
 … code for handling types other than an object …
}
```

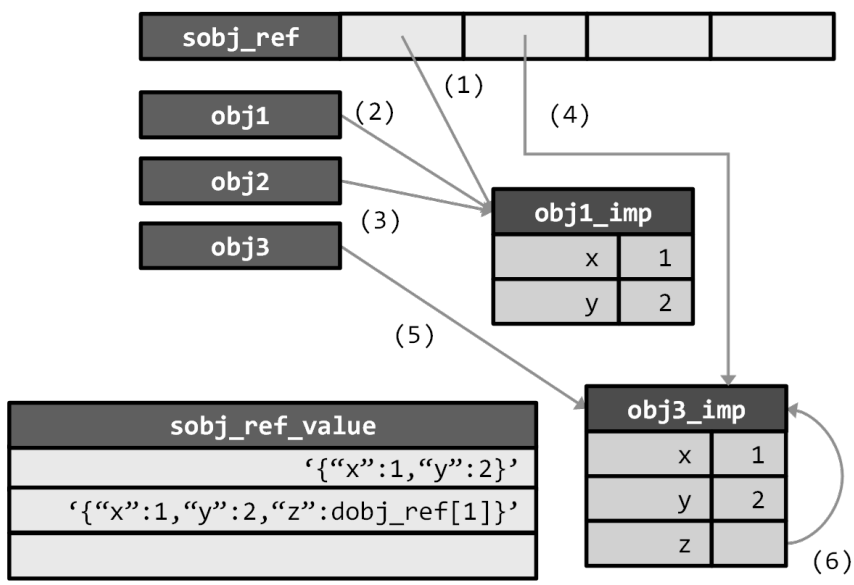


Figure 9. Pseudo code of *JSON.stringify()* and an example using a reference array (*sobj_ref[]*) for Figure 8

array of references, *sobj_ref[]*, to save *unique* objects during serialization. We define the same-sized array, *dobj_ref[]*, in the output JavaScript file to restore only those objects in *sobj_ref[]*, which are then assigned to global variables for correct deserialization.

Figure 9 shows the modified *JSON.stringify()* using an array of references, *sobj_ref[]*. For each object we save its reference in this array, before we serialize/save its JSON string in *sobj_ref_value[]*. If an object reference is already available in the array, the object is not serialized. For the examples in Figure 8, we add the reference *obj1* to *sobj_ref[0]* and serialize/save its JSON string in *sobj_ref_value[0]*. Since the reference *obj2* is already in the array, we do not serialize it. We remember both variables correspond to index 0. For the circular object *obj3*, we add the reference to *sobj_ref[1]*, serialize/save the object in *sobj_ref_value[1]*, and remember the variable is in index 1. During the serialization of *obj3*, we can find that the property *z* is a reference to index 1, so we do not serialize recursively but saves *dobj_ref[1]* instead. This is illustrated in Figure 9. Now, *JSON.stringify()* for each object with an index *i* will return the "*dobj_ref[i]*" string to *state_save()*, which is then saved with its name (*key*) so that "*var key=dobj_ref[i];*" is printed in the output JavaScript file, as explained below.

The *dobj_ref[]* array is first declared in the snapshot JavaScript file as in Figure 10. Each array element is initialized by the object

```
var dobj_ref = new Array();
dobj_ref[0] = JSON.parse('{"x":1, "y":2}');
dobj_ref[1] = JSON.parse('{"x":1, "y":2,
                              "z":dobj_ref[1]}');

var obj1 = dobj_ref[0];
var obj2 = dobj_ref[0];
var obj3 = dobj_ref[1];
```

Figure 10. Snapshot for JavaScript code using *dobj_ref[]*

deserialized from the JSON string saved in the same-index element of *sobj_ref_value[]*. We then generate "*var key=dobj_ref[i];*" for each variable, which will restore the same state of objects.

3.4.2 Saving Closure Function and Closure Variables

JavaScript treats functions as an object (first-class functions), so it allows functions to be passed as an argument or a return value and to be assigned to a variable. It also allows inner functions so that a function can be defined inside another function. A special inner function which accesses the local variables declared in its enclosing outer functions is called a *closure function*, and the outer function's variable is called a *closure variable*.

Figure 11 shows an example. The function *createPerson()* returns an object with *getName()* and *setName()* functions as its properties. These inner functions can access the variable, *name*, declared in the outer function, so they are closure functions and *name* is a closure variable. One issue is that when we invoke *person.getName()* or *person.setName()* after *person=createPerson();* is executed in Figure 11, the variable *name* will be inaccessible since it is a local variable of *createPerson()*,

```
function createPerson() {
   var name = "alice";
   // local variable of outer function
   return {
      getName : function() { return name; },
      setName : function(n) { name = n; }
   };
};

var person = createPerson();
alert(person.getName()); //current name is "alice"
person.setName("bob");
alert(person.getName()); //current name is "bob"
```

Figure 11. An example of closure

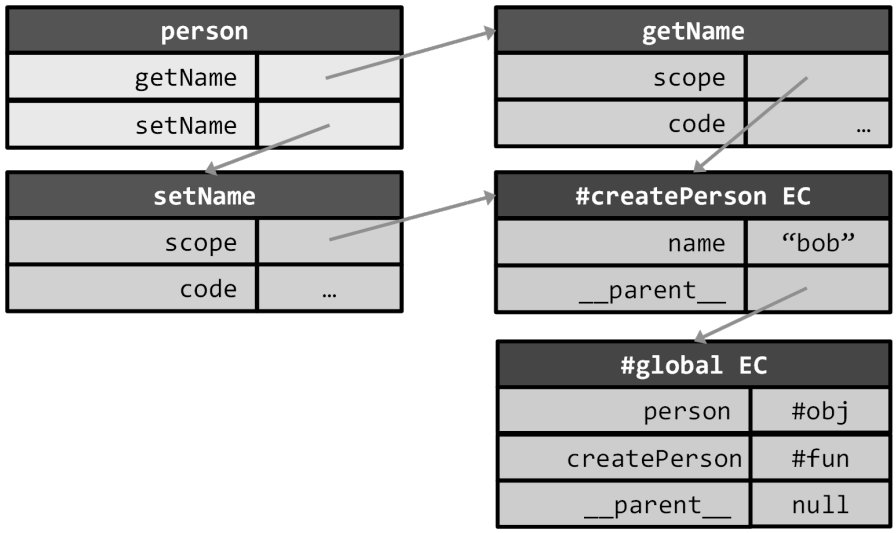


Figure 12. V8 implementation of the closure example

finished by then. So, the JavaScript engine keeps the closure variables even after the outer function finished, by making a scope chain for the closure functions (*getName()* and *setName()*) which includes the outer function’s execution context.

Figure 12 depicts the structure of objects and the scope chain in the V8 JavaScript engine after the JavaScript code in Figure 11 completes. The object *person* has two property objects, *getName()* and *setName()*. Both functions point to the same scope chain via the *context* pointer, where the closure variable, *name*, is accessible.

Now, the problem is how to serialize the *person* object, hence the *getName()* and *setName()* objects. If we invoke *JSON.stringify()* for the global variable *person*, it can serialize *getName()* and *setName()* only, but not the closure variable, *name*. This is so because *JSON.stringify()* can serialize only the properties of an object, while the variable *name* is not a property of the *getName()* or the *setName()* object, but some property of the JavaScript engine. So, the restoration of the app state using deserialization can only recovers

the closure functions, not closure variables, so we need a more elaborate, special way of serialization.

When we serialize a function object, we need to check if it is a closure function. If so, we need to obtain the current values of its closure variables. We also need to check if it shares the same scope chain with other closure functions to see if they have the same outer function instance. Based on these checks, we can identify all closure functions, the value of all closure variables, and their relationship. We added APIs to the browser to access these data.

Now we generate the snapshot JavaScript code for the closures. The only way to restore the closure variables and functions as in the original object state, especially with the scope chain, is simply regenerating them using the outer function call. So, for those closure functions sharing the same scope chain (thus the same closure variables), we construct an outer function object with the closure variables initialized to the current values and make an *anonymous* call to return an object with the closure functions as properties.

```
var dobj_ref = new Array();
dobj_ref[0] = JSON.parse('function() {var name = … }');
var createPerson = dobj_ref[0];

dobj_ref[1] = null;dobj_ref[2] = null;dobj_ref[3] = null;
//references used for closure recovery are initialized to null

var closure1 = function () {
   var name = "bob";
   return {
      func1 : function() { return name; },
      func2 : function(n) { name = n; }
   };
}();
dobj_ref[2] = closure1.func1; dobj_ref[3] = closure1.func2;
dobj_ref[1] = JSON.parse('{"getName" : dobj_ref[2],
                           "setName" : dobj_ref[3]}');
var person = dobj_ref[1];
```

Figure 13. Snapshot JavaScript code for the closure example

Figure 13 shows the JavaScript snapshot code to handle the closure functions and variables in Figure 11. It has an outer, anonymous function with the closure variable *name* initialized to the saved value "*bob*". An anonymous call is made for this outer function, creating an object *closure1* with *func1* and *func2* as its properties. These two closure function objects are saved to *dobj_ref[]* array as previously, and the global variable *person* is restored using these objects. The closure variable is now accessible when the restored closure functions are invoked.

Closures are tricky to save, yet they are used frequently in web apps. In our *tetris* example in Figure 2, an inner function, *gameLoop(),* is defined inside the outer *tetris.play()* function (line 17−26). The outer function defines a variable *me* which is used inside the inner function, so *gameLoop()* is a closure function and *me* is a closure variable. Actually, when *play()* is executed, it registers a timer event with *gameLoop()* as a handler, which will make *gameLoop()* be executed after a given time (*this.speed*). When *gameLoop()* executes, it moves down the current block using the closure variable *me* and then register a timer event again with itself. This makes *gameLoop()* executes repeatedly without a loop, which is a typical way for web apps to invoke a timer event handler repeatedly. We will see shortly how to save *gameLoop()* and register a timer event with it in the snapshot of the Tetris app (see Figure 15).

Actually, closures are used more heavily due to its advantage of *data encapsulation*. That is, many objects that would otherwise be declared as global variables are declared as local variables of an anonymous outer function. When the outer function executes, the local objects become closure variables, residing only in the internal data structure of the JavaScript engine, thus not accessible from the

global variables of the app. Our Tetris app can be programmed in a form of *(function() {var tetris = { … , init=function{..}, play=function{..}}; tetris.init();})();* where *tetris* is now a closure variable (because *gameLoop()* accesses *me* which points to *tetris*), and then there is no global variable at all. If instrumentation were used as in the previous work [15], the *tetris* object would be accessible from some global variables added by instrumentation, compromising the data encapsulation (for example, a third-party widget which shares the same browser session with the instrumented Tetris app can access the *tetris* object via the app's global variables). In fact, most JavaScript framework implement its internal objects by closures. For the jQuery framework, *jQuery* (or $) is the only global object and its internal objects are data-encapsulated using closures [1]. For the Enyo framework, *enyo* is the only global object and all other objects including even the programmer-created objects can be implemented by closures [2]. Instrumentation would expose these framework objects to the outside as well.

3.4.3 DOM Tree and DOM Reference Variable

Everything depicted on the screen during the app execution is a DOM object. In our *tetris* app example, a block made of four rectangle DOM objects is shown with HTML in Figure 1 (line 7-10). We need to save the DOM objects when we save an app state.

Since *JSON.stringify()* can serialize only JavaScript global variables, we save the DOM tree using the JsonML library. JsonML can serialize the DOM objects into the JSON format and deserialize the JSON string to the DOM objects. So, *state_save()* will invoke *JSON.stringify()* for JavaScript global variables and invoke JsonML for the DOM tree. Figure 14 shows the JSON format of a DOM tree.

The tags, attributes, and contents are saved in the JSON format, which is assigned to a JavaScript variable in the *snapshot.js*.

One issue is that some JavaScript global variable (or its properties) may reference a DOM object using a DOM API (e.g., *tetris.controls = document.getElementById("controls")* to point to the control panel in the Tetris app), for interaction between DOM and JavaScript. Since the whole DOM tree is already serialized by JsonML, we do not have to serialize the DOM object referenced by the variable separately. The only requirement is that when the DOM tree is restored, the DOM-reference JavaScript variable should reference a correct, restored DOM object.

To handle this requirement, we perform a preorder traversal of the DOM tree during *state_save()* and save the reference of each DOM object in an array. Then, for each DOM reference variable, we save the index number of the array that matches its reference. Figure 14 shows the preorder index for each node.

When we restore the DOM tree, we also perform a preorder traversal for the restored DOM tree and save the reference of each

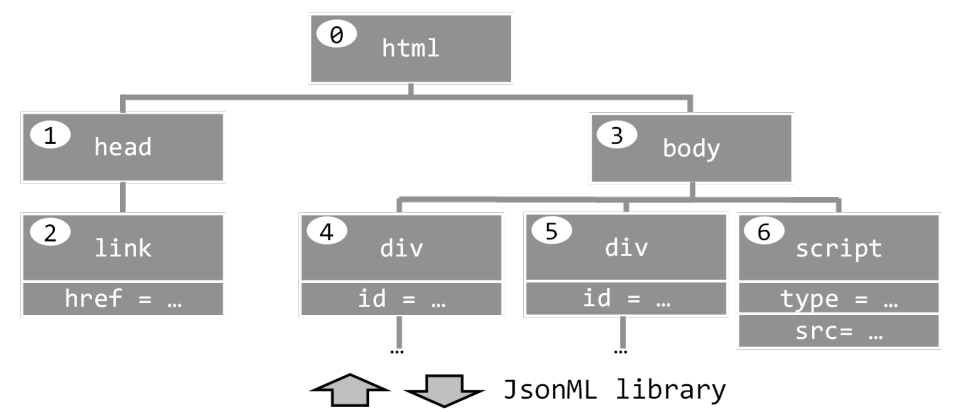


Figure 14. A JsonML example

DOM object in an array (i.e., *jsonML.DOM[]* in Figure 15). Now, each DOM reference variable is initialized by the array element value whose index matches the saved index number of the variable. This traversal and initialization is done by the JavaScript code in the *snapshot.js* (i.e., *jsonML.traverse(document)* in Figure 15).

Some DOM object referenced by a JavaScript variable might not exist in the DOM tree, though. For example, if the *createElement()* DOM API is used in the JavaScript code, the DOM object is created yet is not in the DOM tree unless it is explicitly added by *appendChild()*. There are cases where the DOM object not in the DOM tree can be useful. For example, if an image DOM object is downloaded at runtime and is added to the DOM tree, there might be a delay in displaying the image because of the network traffic. Instead, the image can be accessed and created as a DOM object in the JavaScript code as early as possible to reduce the delay. We need to serialize these DOM objects separately using JsonML in *state_save()*.

3.4.4 Event Handler

After the loading of web apps, event-driven computation proceeds such that event handlers (JavaScript functions) are executed when events occur. For app migration, we need to save the current state of the events and the event handlers, which will then be restored after migration. One problem is that events are not accessible from the *window* object unlike global variables because events are not the properties of the *window* object. Events are actually maintained by the browser in its event queue, so we need to access the browser to find which events are registered and on wait. We re-register those events with the event handlers when we restore the app state.

Some event handlers are implemented by anonymous functions

(e.g., *setTimeout(function() {…}, 1000);*, which makes an event fire after 1000ms and be handled by a function with no name), instead of a JavaScript global function. JavaScript global functions will be serialized when we make a snapshot. However, an anonymous function cannot be accessed by any JavaScript global variable, so we need to access the browser to find those anonymous event handlers and include them in the JavaScript snapshot file (e.g., *dobj_ref[i]=function() {…};*). We also need to register the event again (e.g., *setTimeout(dobj_ref[i], remaining time);*). There are two types of events: DOM event and Timer event.

3.4.4.1 DOM Event

A DOM event is a mouse-click or a keyboard-press event associated with a DOM tree node. For example, when you click a mouse for the *Tetris* app for the control panel, a DOM event associated with the "*controls*" DOM object (referenced by *tetris.controls*) occurs; it will invoke the registered event handler, *cb()*, for rotation or fast downward movement of the current block (it should be noted that *cb()* itself is a local variable which will disappear, so it should be handled as an anonymous event handler as described above).

We can attach a DOM event handler to a DOM tree node in the HTML tag (e.g., *<element onclick = "SomeJavaSciprtCode">*) or in the JavaScript code (e.g., *object.onclick=function() {…};* after obtaining the *object* using a DOM API function as in *object = document.getElementbyId("..");*). This will attach an event to the DOM tree node as an attribute. So, when we serialize the DOM tree, we check if there is an event associated with each node and generate a JavaScript statement to register an event handler (e.g.,

JsonML.DOM[4].onkeydown = dobj_ref[5]; in Figure 15). There is one more way of attaching an event handler in the JavaScript code via *addEventListner()* (e.g., *object.addEventListner("click" , function);*), yet this event handler is not attached to a DOM tree node as an attribute. For finding this type of event handlers and saving them, we made a special API.

3.4.4.2 Timer Event

A timer event is registered using a JavaScript API function. It is either an event that calls the event handler once after a given amount of time (*setTimeout(handler, time)*), or an event that calls the event handler repeatedly on a given interval (*setInterval(handler, interval)*). Registered timer events and their corresponding handlers are maintained by the browser, so they cannot be accessed from the *window* object and no APIs are provided to access them in the JavaScript specification. So, we made a new JavaScript API to be called from *state_save()* when we make a snapshot. The API accesses the current timer event list in the browser to return the current timer events registered, their corresponding event handlers, the event registration time, and the event firing time. In the JavaScript snapshot file, we generate the JavaScript code that registers these timer events, after computing the correct remaining time.

3.4.4.3 AJAX Event

AJAX is for asynchronous communication with the server. We create an XMLHttpRequest (XHR) object, make a server request after registering a call-back function as an event handler, and continue execution. When an answer comes from the server, an event occurs,

executing the call-back function. If we need to migrate the app before the answer comes from the server, there will be a problem. To handle this, we made a new API to collect the current state of all XHR objects, so that we can make all pending requests again after migration and restoration.

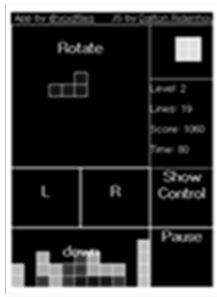
3.5. Implementation

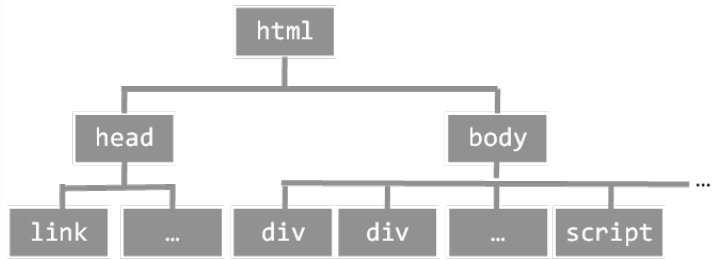
When we press the save button of the extension, the *state_save()* JavaScript function will be executed. Pressing the save button, in actuality, works as a timer event, so it is registered in the event queue of the browser (it is equivalent to executing *setTimeout(state_save(), 0)* with the timeout of zero second). When the browser dequeues the event immediately, it will invoke *state_save()* as an event handler. It should be noted that a JavaScript event handler is executed in a single-threaded manner such that only one JavaScript function is executed at a time during the app execution. So, when *state_save()* is executed, no other JavaScript function is in execution, meaning that neither the JavaScript global variables nor the DOM tree can be changed during the saving, and no event is fired. This will save the app state correctly.

Two files are created as a result of executing *state_save()*: one JavaScript file and one HTML file. The JavaScript file includes the JSON-based state restoration code and the event handlers. The HTML file simply calls the JavaScript file and loads the CSS of the original app. The two files can be stored on the local storage or delivered to the server or the remote device, depending on the purpose of the snapshot.

The DOM tree is saved first after the indexing of DOM-tree nodes based on the preorder traversal is made. Then, the global

Ongoing Game State





variable	value
level	2
lines	19
score	1060
tetris	#obj
...	...

Timer Event

```
{gameLoop() – dobj_ref[1], 135} …
```

DOM Event

```
{cb() – dobj_ref[12], click, 7}
{cb() – dobj_ref[5], keydown, 4}
…
```

Save/Restore

snapshot.html

```
<html>
<head> …
 <script type="text/javascript" src="jsonml.js"></script>
 <script type="text/javascript" src="snapshot.js"></script>
</head> <body> </body>
</html>
```

snapshot.js

```
//for Object & Array Reference
var dobj_ref = new Array();
dobj_ref[0] = JSON.parse('function() {…}');
dobj_ref[1] = null;//initialized to null for closure recovery
dobj_ref[2] = JSON.parse('{"move":dobj_ref[0],
                          "play": …} … ');  …
//for recovery of JavaScript State
var level=2; var lines=19; var score=1060;
var tetris = dobj_ref[2]; …
//for recovery of closure function gameLoop()
dobj_ref[1]=function() {var me = dobj_ref[2]; return
function(){ … }}();
//for recovery of DOM State
var jml = ["HTML",["HEAD" …],["BODY",["DIV",{"id":"tetris"
…}…]["SCRIPT" …] …];
dom = jsonML.toHTML(jml); document.appendChild(dom);
jsonML.traverse(document) //preorder traversal
//for recovery of timer event with gameLoop()
setTimeout(dobj_ref[1], 135);
//for recovery of a DOM onclick event
jsonML.DOM[7].onclick = dobj_ref[12]; //attach DOM event
… …
```

Figure 15. Snapshot JavaScript and HTML for Tetris app

variables of the *window* object is stored in the JSON format, with the DOM reference variables and other variables being saved separately in the file. Other states are saved including the JavaScript code to restore the DOM/time events and their corresponding event handlers.

For state restoration, we simply execute the snapshot HTML file, which, in turn, executes the JavaScript file thru the script tag in the HTML file. The *window* object' s global variables other than the DOM reference variables will be restored first from the JSON format string included in the JavaScript file. Then, the DOM tree will be restored so that the DOM objects are allocated. Preorder traversal of the DOM tree is made to get the reference of the DOM objects, which are then assigned to the DOM reference variables. Finally, the timer events and the DOM events are restored to complete the restoration of the app state.

Figure 15 shows a sketch of the snapshot HTML and JavaScript files for Tetris, saved in the middle of execution.

3.6. Evaluation

This section evaluates the proposed app migration framework, which works correctly for three apps currently.

3.6.1 Experimental Environment

The web apps we used for evaluation are listed in Table 1, with the video clips to illustrate app migration between browsers. They are game apps which manipulate the DOM tree using the event handlers, showing a dynamic behaviour.

We experimented with the X86 desktop and the ARM embedded board. The desktop has the i7-2600 3.4GHz CPU with 4GB memory, and the embedded board has the ARM Cortex-A8 1GHz CPU with

Table 1. Sample web application (video clip at goo.gl/dVF5kZ)

A	Tetris (tetris.alexkessinger.net)
B	Sokoban (www.lutanho.net/play/sokoban.html)
C	Bunny Hunt (www.themaninblue.com/experiment/BunnyHunt)

1GB memory. We use the Chrome browser r128665 with V8 JavaScript engine v3.9.24.

3.6.2 State Saving and Restoration Time Overhead

The app migration would be useful when the state restoring/saving time overhead is reasonable. For each app, we save and restore the app after running it for around 20 seconds. We repeat the experiment 10 times and take the average time overhead (since the exact save point would be slightly different from run to run).

Figure 16 shows the time overhead of each app for the x86 desktop and the ARM board environment. The save time means the execution time of the *state_save()* after pressing the saving button of the extension. The restoration time means the loading time spent for executing the HTML file. The time overhead even for embedded

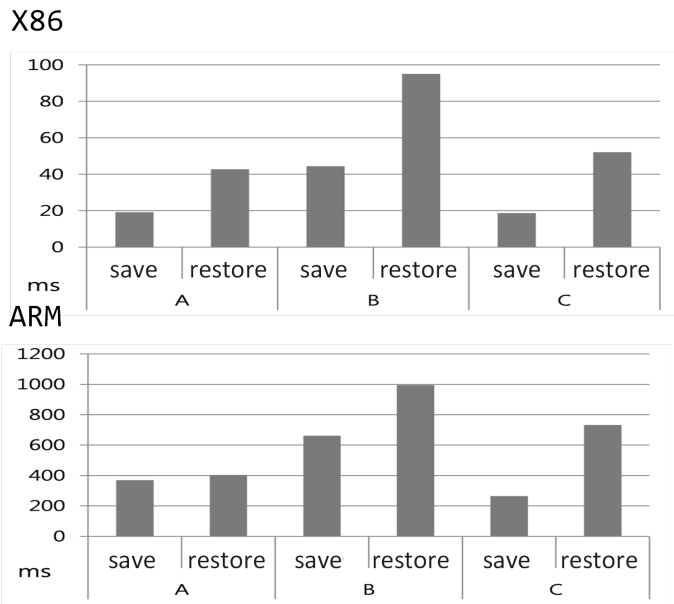


Figure 16. Save/restore time (ms) of migration framework

board is within one second, which do not make users feel a big overhead.

3.6.3 Snapshot File Size

Figure 17 shows the file size of the snapshot for each app and the distribution of the HTML, CSS, and JavaScript portions, compared to the original app. The HTML tags in the app source file are reduced to a single HTML tag to invoke the JS file in the snapshot, as shown in Figure 15, so the HTML portion is minimal.

The JavaScript portion increases significantly since it now includes the DOM tree in the JSON format. For the same DOM tree, the size of the HTML tags and the size of the JSON string would be similar (less than 5% difference in size). As the app execution proceeds, however, the DOM tree can be expanded as more nodes are added by the JavaScript code execution, so the corresponding JSON string is larger than the original HTML tags. Figure 17 shows that the JavaScript portion generally increases more than the reduced portion of the original HTML tags due to the expanded DOM tree.

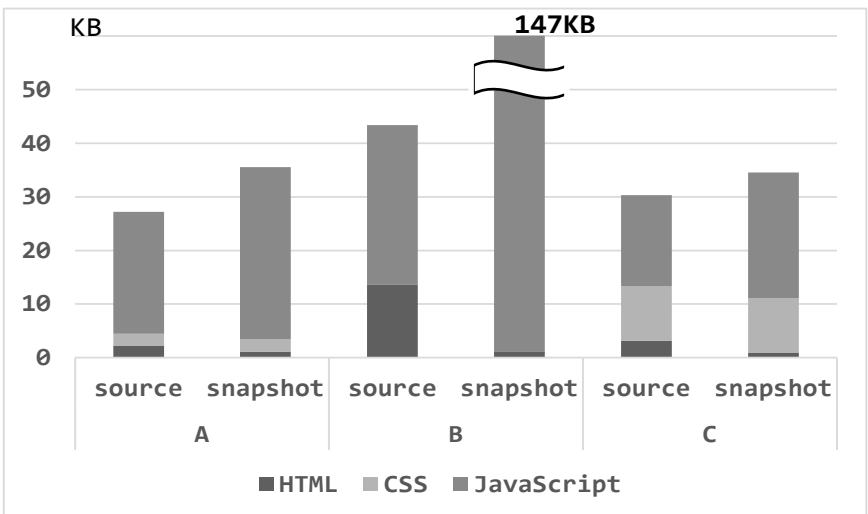


Figure 17. Size (KB) of app source and snapshot files

The increase is pronounced for the B app since its DOM objects were added significantly by the time when we take a snapshot.

3.6.4 Portion of State Save/Restoration

Figure 18 shows the distribution of the saving time (execution time of the *state_save()*) among the saving of the DOM tree, the saving of the JavaScript variables, and the saving of others (etc.) which includes the saving of the events, the event handlers, and the file I/O time. A larger DOM tree in the B app also makes the DOM saving time larger than the saving time of the JavaScript variables.

Figure 19 shows the same distribution of the restoration time of the app state from the snapshot. The etc. portion in Figure 19 also includes the file I/O time which is much smaller than that in Figure 18 because the file write takes much longer than the file read. One

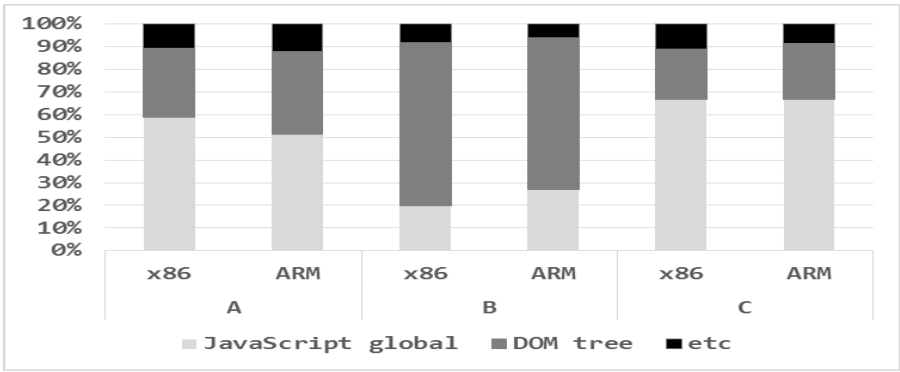


Figure 18. Saving time distribution

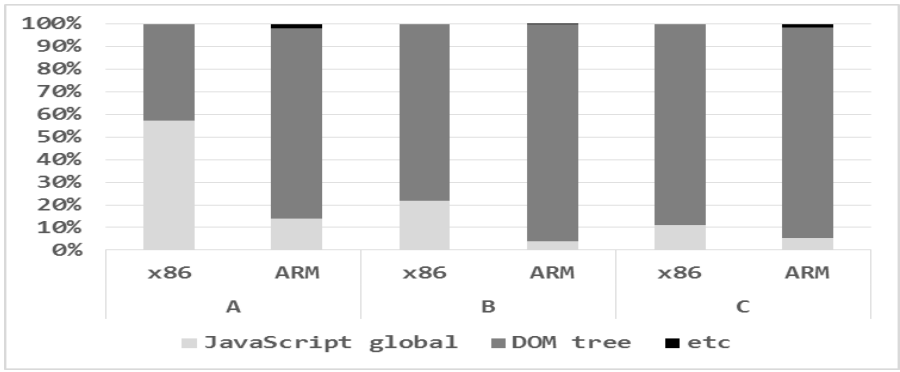


Figure 19. Restoration time distribution

thing to note is that for the DOM objects the ARM restoration time is larger than the x86 restoration time. DOM restoration requires the invocation of the DOM and JsonML APIs, and it appears that the ARM Chrome browser is less optimized than the x86 Chrome browser. The restoration of the DOM tree of the C app takes a larger portion than the saving, and the C app displays image files unlike others, which might affect the DOM restoration overhead.

3.6.5 Number of Objects Saved and Restored

Table 2 shows the number of objects saved and restored for each app, for each type of objects discussed in this paper. It shows the number of JavaScript global objects created by the programmer (a). The number of closure functions is also shown (b). The number of DOM objects in the DOM tree (c) and the number of DOM reference variables (d) are also included, with the number of DOM objects not included in the DOM tree (e). Finally, the table shows the number of the timer events (f) and the DOM events (g) when we make a snapshot, which is the same as the number of event handlers saved. We did not see an example of circular references, but we saw a reference alias problem in the C app.

Our sample apps do not use a web framework such as jQuery [1], Enyo [2], or Ext JS [3]. If an app is programmed using a framework, many API function objects will be created at the framework initialization time. We can save these objects in the snapshot with other app objects and migrate (this would work since we can save

Table 2. Number of Objects Saved

	(a)	(b)	(c)	(d)	(e)	(f)	(g)
A	109	4	59	23	0	2	2
B	61	16	786	0	14	0	17
C	35	4	56	4	16	7	5

the framework objects in the binary form and restore [12]). Or, since most framework objects are functions unlikely to be changed by the programmer, we can save the minimum objects but invoke the framework initialization in the snapshot to re-create most framework objects at the target (as we do for JavaScript built-in objects which are not saved in the snapshot file but re-created on the target JavaScript engine when it is initialized). We might need to handle cases when the programmer adds his own APIs to the framework or when the APIs are invoked for a DOM object accessed via the framework. These are left as a future work.

3.7. Other Usage of the Snapshot

As mentioned before, our snapshot is a complete web app in itself. Compared to the original app, it is another app whose execution can restore the saved state and continue its execution. We can exploit the snapshot for other purposes.

The snapshot can be used for accelerating the app *launch time*. If the same job is repeated during the app launch time (e.g., initialization of the web framework such as jQuery), we can save the launched state as a snapshot and start from the snapshot when launching an app to reduce the launch time. For faster restoration, however, the JavaScript objects would better be saved in a binary form as they are in the heap, rather than the JSON-based text form; the binary form of objects just need to be copied and relocated to the heap, while they need to be created by the execution of the JSON-based JavaScript code. We actually implemented this idea and found that app launch time can be reduced by 20% [12].

A snapshot can be used as a swap space in the web platform. That is, if there are too many apps running on a browser, the oldest

app is forced to be deleted from the job queue, losing its current state. Instead, we can make a snapshot and save for the deleted job. When the job is restarted, its last state can be restored from the snapshot.

When we distribute a complex app (e.g., rich internet application), we can deliver it in a snapshot form after fully setting-up based on the customer's request. This can free the customer from time-consuming, complicated set-up process.

Web browsers save the URL for the bookmark or the history, so any dynamic change made for a web page is lost when we revisit it via the bookmark. If we save the snapshot instead, we can restore the exact same page when we revisit.

Another use of the snapshot is a new-type of app that can be *shared* and executed by multiple users. For example, when a group of people want to schedule a meeting, instead of using a scheduling site such as doodle.com, one person runs a scheduling app to mark his schedule and passes its snapshot to the next person, who then runs the snapshot to mark his and passes. In this way, an app in execution is circulated among multiple users, leading to a new collaborative user experience.

Finally, the snapshot can be used for *off-loading*. If an app includes some time-consuming computations for an event handler, we take a snapshot just before handling the event and send the snapshot to the server for faster execution. We take a snapshot again after the execution and send it back to the client for continuous execution. Our snapshot would make the communication and coordination overhead smaller than the one used in the Android platform [31].

The last two usages will generate the snapshot repeatedly, so the instrumentation approach would be too costly.

Chapter 4. Loading Time Acceleration of Web Applications

4.1. Acceleration of Web Apps with Snapshot

Any web designers can easily develop web apps, because app programming is based on the popular web technology. Moreover, web apps are distributed in the source code format, runnable on any devices where a web browser is installed, allowing one-source, multi-platform environment.

Despite the advantage in portability and productivity, web apps are involved with some performance issues, mostly due to JavaScript. To improve the performance loss on JavaScript interpretation, just-in-time compilation (JITC) is used to compile JavaScript code to machine code at runtime. In fact, when we turn on JITC for web apps, we rarely see any performance improvement.

We propose a different approach to improve the performance of web app. As in other apps, the execution of a web app starts with *app loading*. The loading process includes the HTML parsing and the execution of the global JavaScript code, which initializes the app and generates the DOM tree for the first screen of the app. During this step, JavaScript event handler functions are also registered, which will be executed later when the DOM event or timer event occurs. We focus on accelerating app loading, especially the JavaScript execution. That is, if the same JavaScript code is executed during every app loading, it would be better for performance to save the JavaScript execution state and start the app from the saved state. In fact, web apps are often programmed using a *web framework* such as *jQuery* [1], *Enyo* [2], or *Ext JS* [3], and the same JavaScript code is

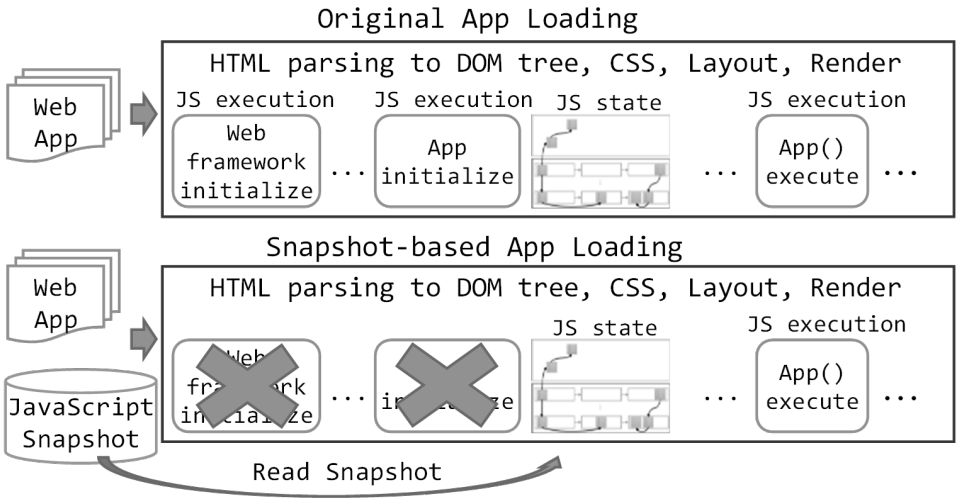


Figure 20. Original vs. snapshot-based app loading

executed during app loading to create and initialize many framework objects. Also, app-specific objects are created during app loading. If we save the initialized state of these objects in the form of a *snapshot* where they are saved in a binary form as they are in the heap, and start app loading from the snapshot, we can shorten app loading process. Figure 20 illustrates the proposed idea. We skip some of the JavaScript execution during app loading using the snapshot.

We implemented the snapshot for the JavaScriptCore engine and experimented on an ARM embedded board. We could observe a tangible (20%) difference for the whole app loading time for Enyo apps. The Enyo initialization time is reduced by 75%. We also experimented with the jQuery and other frameworks, and their initialization time is also reduced similarly. We also see a similar reduction of initialization time on x86.

4.2. Saving the JavaScript State

The JavaScript engine must maintain many data during the execution of a script tag, including all the JavaScript global objects in the GEC and all the local objects in the EC chain. If we make a snapshot in the

middle of execution for a script tag, all of these data should be saved, causing an issue of space overhead. More seriously, restarting the JavaScript execution in the EC stack would be tricky, complicating the restoration process.

To reduce the space overhead and restoration complexity, it is better to make a snapshot *after* the execution of a script tag because at that point the JavaScript engine must keep only the global objects in the GEC. Also, no JavaScript code is in execution, so we do not worry about restarting the execution in the middle of the JavaScript code. Other data to keep after a script tag are DOM events and timer events, which are accessible from the JavaScript *window* object, thus being saved easily with other objects (see Section 4.5.4).

Now we discuss after which script tag we can make a snapshot. In our Tetris example in Figure 2, there is one big difference between line 1~32 and line 33, regarding the saving and restoring of the app loading state. The line 1~32 simply defines JavaScript variables and functions, without affecting the DOM state or even the event state. That is, the line 1~32 defines only the function objects that update the DOM state or the event state (*init()*), without executing it. Actually, the line 33 executes these functions to change the DOM tree or to register an event handler. This means that the execution of the line 1~32 can affect only the JavaScript state, while the execution of the line 33 affects the DOM state and the event state as well.

Affecting the event state is not a problem since we can restore the event state from the saved events. However, if a script tag changes the DOM tree, we need to save the DOM tree to restore the same loading state, which is complex as discussed before. So, if we make a snapshot right after the first tag, and restore the JavaScript

state directly from the snapshot without executing the first tag, we can reduce the loading time, with the same loading state of the original one.

There is an important script tag often included in a web app HTML file, whose execution affects only the JavaScript state. It is called a JavaScript *web framework*.

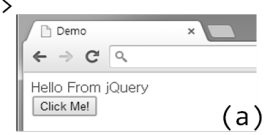
4.3. JavaScript Web Framework

Web apps often employ a *web framework*, a cross-platform JavaScript library, for faster app development [32]. Popular web frameworks are jQuery [1], Enyo [2], and Ext JS [3]. A web framework, written completely in JavaScript, is included in the HTML file with a script tag (e.g., *<script src= "jQuery.js" ></script>*) as other JavaScript code is. It is often included as the first script tag and executed to initialize the framework, creating many JavaScript (function) objects, which will be used by other JavaScript code. The framework JavaScript code will never affect the DOM state, yet its execution time is substantial, so it is desirable to make a snapshot right after the framework script tag. For Enyo and Ext JS frameworks, the app JavaScript code itself (*app.js*) is made by a build process (*deployment* in Enyo and *packaging* in Ext JS), and is added to the HTML file via another script tag. This app script tag does not affect the DOM state either, so it is better to make a snapshot right after the app script tag rather than the framework tag to save both the framework and the app JavaScript state, for even faster app loading (see Section 4.4).

Figure 21 shows a jQuery framework and app example. The jQuery library is included in the HTML file via <script src = "jQuery.js" ></script>. Then, the app can be programmed using the

jQuery App

```
1 <!doctype html>
2 <html><head><meta charset="utf-8" />
3  <title>Demo</title></head>
4 <body> <div>Hello From jQuery</div>
5  <button type="button">Click Me!</button>
6  <script src="jQuery.js"></script>
7  <script src="app.js"></script>
8 </body> </html>
```



(a)

jQuery.js

```
1 (function(…) { … var jQuery = function(…) {…}  …
2  jQuery.fn = jQuery.prototype = { jquery: version, … }
3  jQuery.extend = function() {…}
4  jQuery.event = { … click: { … }  }
5  window.$ = jQuery;
6  …
```

(b)

app.js

```
1   $( "button" ).click(function() {
2     $( "div" ).css("color", "red");
3   });
```

(c)

Figure 21. jQuery framework and an example app

selector with the jQuery object (also designated by $), and many property functions of $. For example in Figure 21 (c), we can select a DOM node for the anchor element using *$("button")* and register a *onclick* event handler using the *click()* function in line 1 such that the color of the text on screen is changed to red.

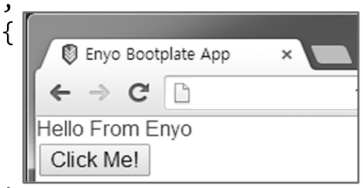
Figure 22 (b) shows a simplified Enyo framework source code (*enyo.js*). The execution of the *enyo.js* creates a single global object, *enyo*, and its property API functions to be used by the Enyo app. Figure 22 (c) shows an example of an Enyo app. Enyo provides a constructor function called *kind()*, as a *class* in Java. For example, *app.js* defines its own kind, using an existing kind, *Button*, in line 3. If the user clicks the button on the screen, the event handler in line

index.html

```
1  <!DOCTYPE html>
2  <html>
3   <head> <title>Enyo Bootplate App</title>
4    <link href="enyo.css" rel="stylesheet"/>
5    <link href="app.css" rel="stylesheet"/>
6    <script src="enyo.js"></script>
7    <script src="app.js"></script>
8   </head>
9   <body class="enyo-unselectable"> <script>
10    new HelloWidget().renderInto(document.body);</script>
11  </body>
12 </html>                                                    (a)
```

enyo.js

```
1  (function(){ var e = "enyo.js";
2    enyo = window.enyo||{options:{}},
3    enyo.locateScript = function(e) {
4    … }; enyo.args=enyo.args||{};
5    …})();
6  enyo.version = {enyo:"2.4.0"};
7  (function(e){
8    var t = [], n = function(n){ … };
9    e.rendered = function(e,n) {t.push([e,n])},
10   e.addToRoots = function(t) { … }
11 })(enyo);
12 …                                                          (b)
```



app.js

```
1  enyo.kind({name: "HelloWidget",
2   components: [{name: "hello", content: "Hello From Enyo"},
3   {kind: "Button", content: "Click Me!", ontap:"helloTap"}],
4   helloTap: function() {
5    this.$.hello.applyStyle("color", "red"); }});          (c)
```

Figure 22. enyo framework and an example app

3 will be executed and changes the colour to red in line 5, which will be displayed on the screen. It should be noted that *app.js* simply defines a kind and an event handler, yet it does not execute any event handler that changes the DOM tree.

To estimate if the framework snapshot will be useful, we evaluated the web frameworks on an ARM embedded board. We experiment with an app similar to the one in Figure 21 and Figure 22 for each framework. The framework initialization time, its portion in

Table 3. Framework overhead on Pandaboard ES and WebKit

	Initialize time	Ratio	File size
jQuery 1.11	84 ms	23.4%	95 KB
Enyo 2.3.0	309 ms	49.9%	611 KB
AngularJS 1.3.14	71 ms	18.0%	123 KB
MooTools 1.5.1	128 ms	32.7%	152 KB
PrototypeJS 1.7.2	103 ms	30.3%	194 KB

app loading time, and the file size of its JavaScript code are listed in Table 3. jQuery is lighter than enyo, yet its loading time takes 23% of the app loading time. Enyo is much more substantial, taking 50% of the loading time, respectively. So, the framework snapshot is expected to reduce the app loading time significantly.

4.4. Approach to Taking the Snapshot

This section describes a more detailed scenario and approach to exploiting them.

Our snapshot will include all JavaScript objects starting from the *window* object (GEC), live when we take a snapshot. Events registered by JavaScript execution are also included. The DOM tree is not saved in the snapshot, but the *document* JavaScript object that points the root of the DOM tree is a property of the GEC, thus being included as other objects.

When we run the same app with the snapshot, HTML parsing will make a DOM tree as usual, which will be the same as the previous one, yet newly created. The *window* object is not created yet. When we encounter the snapshot, we will restore all JavaScript objects based on the snapshot, including the GEC, which we will now make the *window* object of the app. The *document* object will also be restored, yet will now point the root of the newly created DOM tree.

Above scenario implies two conditions. First, there must be a single snapshot since we cannot make multiple *window* objects from multiple snapshots. Also, the first script tag should be included in the snapshot; otherwise there should be no snapshot. This is because it is impossible to create a *window* object from the first script tag not included in the snapshot, then encounter a snapshot and create another one.

Making a merged snapshot from multiple script tags including the first one would be desirable for better impact of the snapshot, and the merged snapshot is made (and restored) at the end of the last script tag. Obviously, they should be continuous; otherwise a middle script tag not included in the snapshot would violate the execution order.

A script tag cannot be included in the snapshot if it changes the DOM tree as we discussed in Section 2; since a DOM tree is not saved, such a change cannot be restored from the snapshot. Unfortunately, it is not always simple to automatically decide if JavaScript code affects the DOM tree. Therefore, we assume that the app developer specifies explicitly if a script tag affects the DOM tree. For this we added an attribute *snapshot* to the script tag such that if

```
1  <!doctype html>
2  <html>
3  <head>
4   <title>App</title>
5   <script src="A.js", snapshot=true></script>
6  </head>
7  <body>
8   <script src="B.js", snapshot=true></script>
9   <script src="C.js"></script>
10  <script src="D.js", snapshot=true></script>
11 </body>
12 </html>
```

Figure 23. An example of HTML file with snapshot attribute

snapshot=true is specified, the corresponding script tag does not affect the DOM tree, thus appropriate to be included in the snapshot; If it is missing, the tag cannot be included.

Consider an HTML file in Figure 23. Notice that script tags on line 5, 8, and 10 have snapshot attribute, while the one on line 9 does not. So, the point we take the snapshot will be at the end of line 8, because line 9 does not have snapshot attribute, which makes line 10 not be included in the snapshot. Notice that HTML tags can be freely located between any two snapshot script tags, as line 6-7.

Since JavaScript code in the web app can be modified, the snapshot should be validated before being used. This can be checked by comparing the app source file. If a new JavaScript file is downloaded, we simply remove the snapshot, which makes the next run of the app generate a new snapshot automatically, then used thereafter for the next runs. For install-based web platforms there will be a version control, so we control the snapshot based on the app version.

4.5. Saving and Restoring the Snapshot

The previous section described our approach to saving and restoring a snapshot. This section describes detailed techniques on how to save the JavaScript state to a snapshot and how to restore it from the snapshot.

4.5.1 Approach to Save/Restore JavaScript State

The JavaScript state is composed of the JavaScript objects in the heap starting from the GEC, so we need to save them to the snapshot file. For faster restoration, we save them in a binary form as they are in the heap so that we can simply copy the objects from the snapshot

back to the heap with relocation. One issue is that the heap is shared by multiple apps, so the heap is mixed by objects created by different apps. This means that we cannot save and restore the whole memory image of the heap at once but save and restore the objects of an app one-by-one. Another issue is that the JavaScript state is not entirely composed of the JavaScript objects but the DOM objects that reference the DOM elements created by the browser. So we need to save the information on these references to the snapshot and restore them for the new DOM elements created during app loading.

How objects are implemented in the heap depends on the JavaScript engine. Our development is based on the JavaScriptCore (JSC) engine of the WebKit browser [33] and V8 engine of the Blink browser [34], so we first describe the JSC snapshot, followed by the V8 snapshot.

4.5.2 Snapshot for JSC

JSC does not include any support to identify and save the objects in the heap, so we need to implement the saving module on our own (on the other hand, V8 has a limited support of saving the heap called a serializer, which we enhanced as will be seen in Section 4.5.3.4). We save the binary form of each object, added with some information needed for restoration.

4.5.2.1 JavaScriptCore (JSC) Heaps

Figure 24 depicts the objects in the JSC heap and illustrates how they are saved to the snapshot file and restored from the snapshot to the heap. When we make a snapshot, the GEC (the *window* object) constitutes the only JavaScript state. As we explained before, the window object has many properties. One important property is the

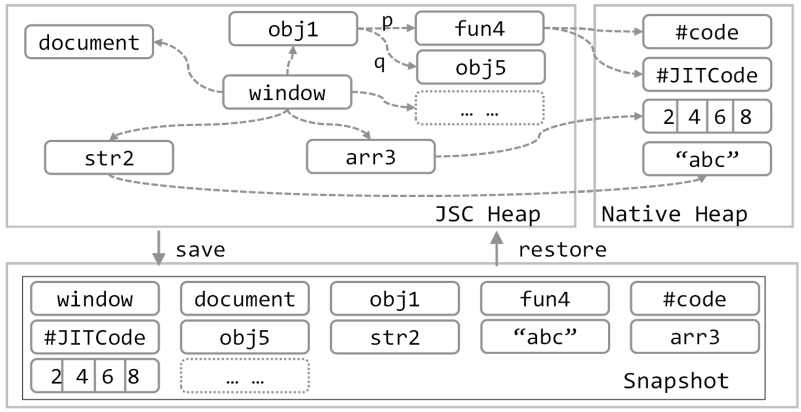


Figure 24. Objects in the JSC heaps and snapshot

document object. There are also built-in objects such as *Math* or *Navigator* objects created at the JSC initialization. The objects created by the JavaScript framework are properties of the *window* object as well. Finally, the global objects created through the JavaScript global code by the programmer are also properties. These objects can also have properties that point to other objects. Consequently, all the objects form a *rooted graph* where the *window* object is the root, as depicted in Figure 24.

There are two types of heaps used in the JSC to allocate objects. The first one is the JavaScript heap of the JSC engine, which we call the *JSC-heap*. The other is the heap of the browser, which we call the *native-heap*. Most objects are allocated to the JSC-heap, but some objects such as the characters of a String object, or the source code, bytecode, and JIT-compiled code of a function object are allocated to the native-heap. Many objects in the JSC-heap use a hash table data structure which also resides in the native-heap. A DOM tree node pointed by a JavaScript object is in the native-heap since the browser maintains the DOM tree. The object in the JSC-heap is garbage-collected by the JSC engine, while the object in the native-heap has a reference counter, and is reclaimed when all their

parent objects in the JSC-heap become a garbage (some string data in the native-heap can be pointed both by the JSC engine and by the browser, so the reference counter should be checked before being removed). It is also possible for some hash table in the native-heap to point an object in the JSC-heap, which complicates the object graph. So, we should understand the object relationship clearly to save the objects correctly. When we restore, we also need to allocate each object to its original heap (JSC or native) properly. This requires an elaborate saving and restoring technique.

4.5.2.2 Objects in the Heap in JSC

There are many different types of JavaScript objects and other objects created during JavaScript execution, which are implemented differently.

JavaScript data types include String, Object, Number, Boolean, Array, Null, and Undefined. In addition to these typed objects, built-in objects such as Function, Date, RegExp, and etc. are also valid JavaScript objects. To manipulate the DOM tree, DOM objects are also required.

The most basic JavaScript object is the generic *Object* with properties and values. Figure 25 shows an example of an Object *objA* with two properties *x* and *y*. It also shows how *objA* is represented in the JSC-heap in a simplified form. Every JavaScript object in JSC has a descriptor object called a *structure* (called a hidden class in the V8 engine), which includes the internal property structure of the object (*property table*) and the prototype object (*__proto__*).

The property table has the offset of each property, used to access the *storage* of the object. For example, the access to *objA.y* should first find the offset of *y* from the property table, which is then

added to the start address of the storage. These structure, property table, and storage are not JavaScript objects but *helper* objects used to implement the JavaScript object. Both the property table and the storage are in the native-heap, as depicted by the dotted rectangles in Figure 25.

The prototype object is a JavaScript object, used to search for a property if it does not exist in the storage of an object; _proto_ pointers are repeatedly chased until the property is found, as what inheritance does in other languages.

Figure 25 also shows the implementation of a String object. It also has a structure object, yet the property table is empty since there is no property for a String object. Instead, it has a length and a value for the character string, which is saved in the native-heap, whereas the String object, structure, and prototype exist in the JSC-heap.

Figure 26 shows a function object which also has a structure with an empty property table and storage. Instead, there is an executable

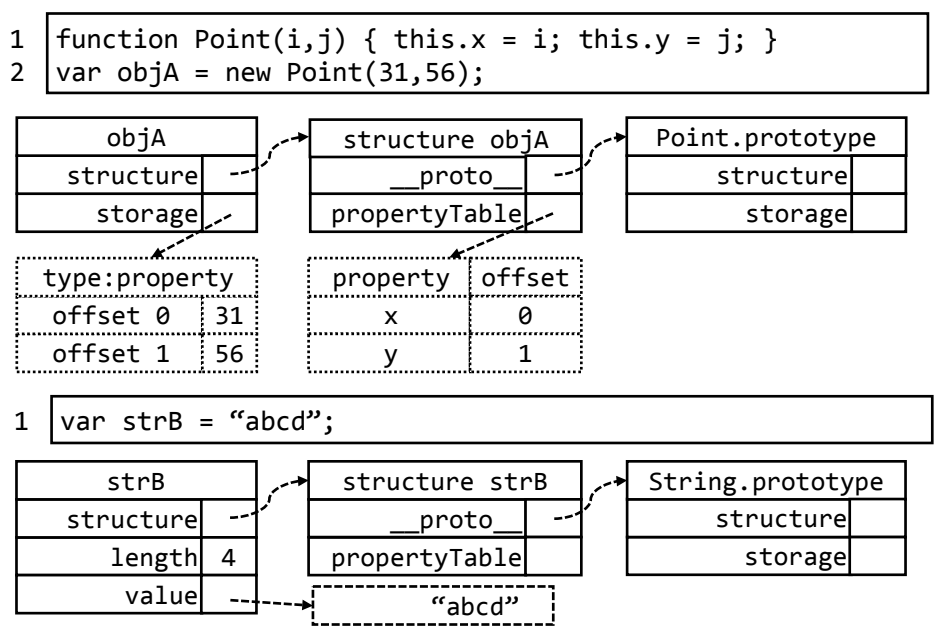


Figure 25. Example of Object and String objects in JSC

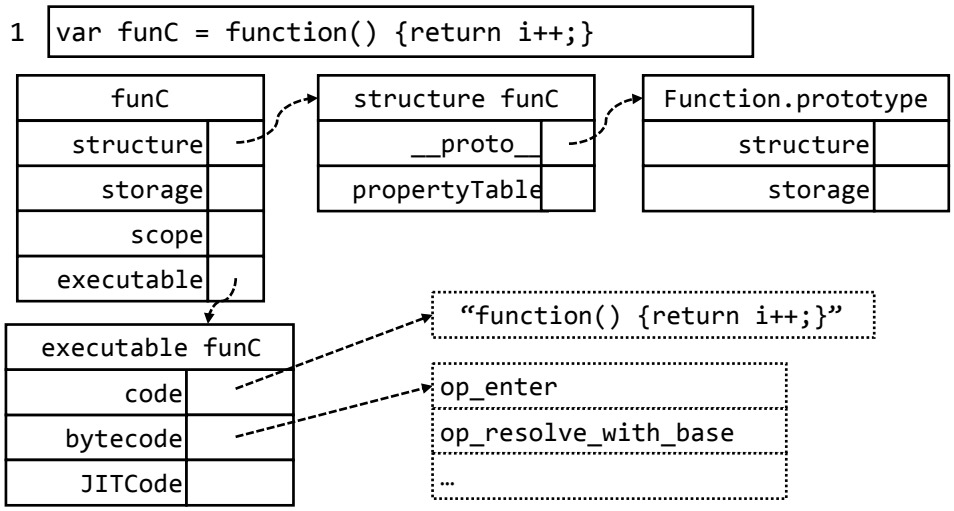


Figure 26. Example of Function Object in JSC

object to keep the source code, which will be saved in the native-heap. Since JSC can use the interpreter and the JIT compiler, both the bytecode and JIT-compiled machine code can exist, which will also be in the native-heap. For the closure functions, the scope chain exists, where the ECs of enclosing functions when the closure function is created are available, making the closure variables accessible.

4.5.2.3 Object Graph Traversal

Starting from the *window* object, the root of the object graph, we need to visit each object, save it to the snapshot file, and visit its property objects. We employ the marking routine of JSC's mark-and-sweep garbage collector (GC) to traverse the object graph in the JSC-heap. When we visit an object, we copy the object to the snapshot file. Then, we mark its unmarked property objects and push on the stack (marked property objects will already be in the stack). We pop and visit the top object on the stack and repeat the same until the stack becomes empty (so this is a depth-first traversal).

4.5.2.4 Object Save

We now discuss how to save a visited object to the snapshot file. The format to save an object depends on the type of the object. Also, since the traversal based on the marking routine visits only the object in the JSC-heap, we additionally need to save the objects in the native-heap. Fortunately, all the native-heap objects are properties of some JSC-heap objects, so we can save a native-heap object easily when we save its parent object in the JSC-heap.

Figure 27 shows the format used to save in the snapshot file, for some of the JSC-heap objects and the native-heap objects. The first field always saves the address of the object. This address should be relocated when we restore the objects from the snapshot, which will be discussed shortly.

For the JSC-heap object, the next field is the address of the JSC's structure called *ClassInfo*, a static constant member of the corresponding JSC C++ class for the object and has the information on the C++ class. So the address for the *ClassInfo* can serve as an identifier for the object type. It includes the class name, the parent class, and the method table for some of the class' *static* method functions such as allocation, de-allocation, visit of property objects, etc. GC, for example, can invoke a de-allocation function for a

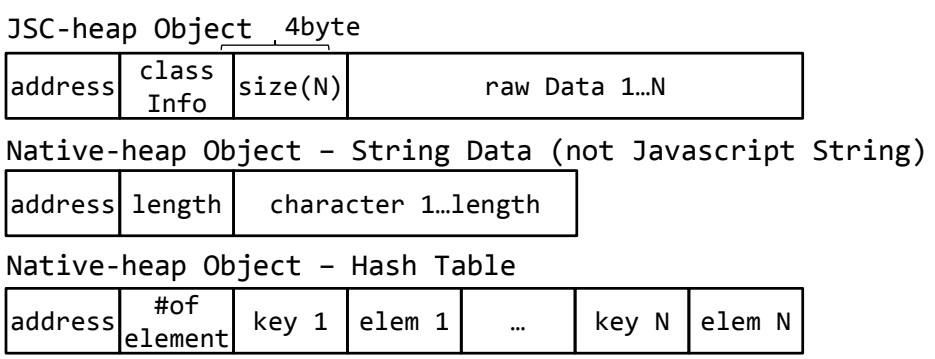


Figure 27. Object snapshot format

garbage object based on the object's *ClassInfo* and its method table, without even knowing the type of the object.

For our snapshot implementation, we added new functions in the *ClassInfo* for saving and restoring. It should be noted that the address of *ClassInfo* does not change between saving and restoring time since it is declared as a static variable. Finally, the size and the raw data of the object are saved.

Native-heap objects are not traversed by our marking algorithm, so they are saved (if they are not saved yet, which can be checked using a table) when their parent JSC-heap objects are traversed. The first field of the saved native-heap object is also its address. Unlike JSC-heap object, we are able to know the type of a native-heap object through its parent JSC-heap object, so we do not save its type information. Then, we save the minimum data (not the whole raw data as in JSC-objects) enough to restore the native-heap object. For example, a native-heap object in a String JSC-heap object contains a stream of characters, length, reference count, and

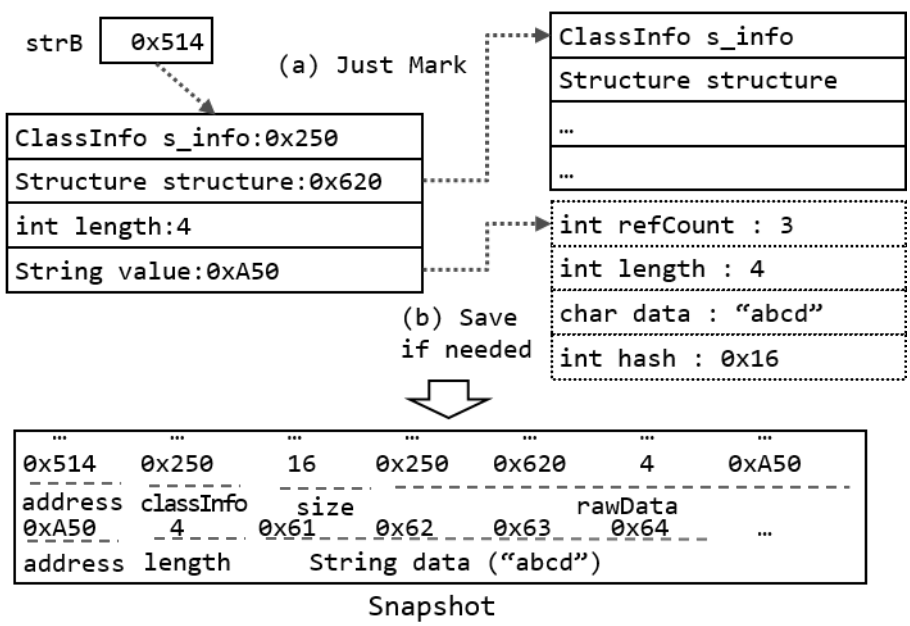


Figure 28. Object save example

hash value, but we only save the stream and its length as Figure 27 in (since we do not have to restore the reference count and the hash value precisely, as will be explained shortly). For a hash table native object, we do not save the whole hash table, but only the key and value pairs existing in the table.

Figure 28 shows how the String object in Figure 25 is saved in the snapshot. The address of the object (0x514) is saved as the first field of the snapshot. The *ClassInfo* address (0x250) is saved next. Then the size of the object (16) is saved, followed by the 16-byte raw data. The object pointed by the *structure* field is a JSC-heap object, thus being simply marked and pushed on the stack, so that it can be saved when it is visited later. The object pointed by the *value* field is a native-heap object, so it is saved now. We simply save the string length (4), followed by the four characters of the string. There are two more data in the value object. The reference count (*refCount*) indicates how many objects point this value object, which is invalid when the value object is restored, thus not being saved. The hash value (*hash*) is used when the string is used as a key for a hash table, which can be regenerated when we restore the object and add it to a hash table, so we do not save it in the snapshot.

4.5.2.5 Object Restore

When we restore from the snapshot, we read each object from the snapshot file one-by-one and restore it to the heap with relocation. The two types of objects, JSC-heap objects and native-heap objects, are restored differently.

When we restore a JSC-object, we first need to allocate a space in the JSC-heap depending on the object type. The *ClassInfo* address saved as the second field of the saved object can be used for this

purpose such that the allocation function in the method table of the *ClassInfo* structure is invoked to allocate the space (we do not have to check the object type specifically since the invocation will call the corresponding function in the object's C++ class). Then, we simply copy the saved raw data to the allocated space using *memcpy* with the size as an argument. This is more efficient than using the constructor function of the C++ class for the object, because space allocation followed by memory copy of the whole chunk would be faster.

When we restore a native-heap object, we cannot use *memcpy* since we saved only the minimum data for the native-heap object. Instead, we invoke the constructor of the corresponding C++ class to allocate the object. Since we read the parent JSC-heap object first before the native-heap object in the snapshot file, we know the type of the native-heap object from the parent object, so we invoke the corresponding constructor to create the object and record the address to the parent object's property. For the string native-heap object in Figure 28, for example, we invoke the constructor of the string class (defined in the WebKit browser) with the length and the character stream only. If the native object is a hash table, we will create a hash table first and enter those key and value pairs to the table for restoration.

There is one important issue for the restoration. The object we allocate in the heap would have a different address from the one we saved in the snapshot file. So we need to *relocate* all the addresses saved to those of newly allocated objects. For this, we make a relocation table during the restoration. One exception is the address (or the offset in the shared library) of *ClassInfo*, which does not

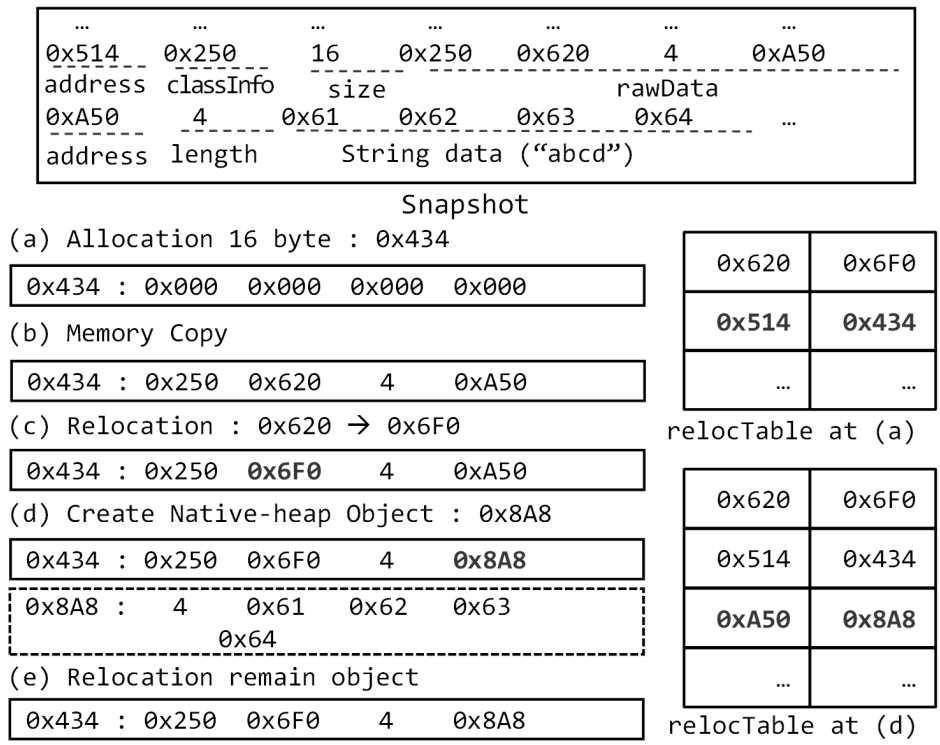


Figure 29. Object restoration example

change between when we save and when we restore, as we mentioned before.

Figure 29 shows the restoration process of the String object saved in the snapshot in Figure 28. We first read the *ClassInfo* address (0x250) and invoke the allocation function using it, as *(0x250)->methodTable()->alloc();*, which will allocate a space for the object in the JSC-heap as in Figure 29 (a). The pair of old address (0x514) and the new address (0x434) is added to the relocation table so that the new address can be used for the restoration of the other objects. Then, we copy the raw data in the snapshot to the allocated space using *memcpy* in Figure 29 (b). Among the raw data, we check if an address of a JSC-heap object is in the relocation table, meaning that the object in the address has already been restored. If so we replace it by the new address. In

Figure 29 (c), the address of the structure object (0x620) is in the relocation table, so we replace it by the new address (0x6F0) since the structure object must have already been restored and allocated. If the address were not in the relocation table, we need to replace this old address later after the object is allocated, so we make a note for this address.

We now restore the native-heap object. If there is no match in the relocation table, we need to create the native-heap object, using the class constructor of the object with the data in the snapshot. In Figure 29 (d), we create a new native-heap object for the string (with the length and the character stream only) whose address is 0x8A8, so the old address (0xA50) is replaced by the new address, and the address pair is added to the relocation table. If we need to create a hash table, there can be a case where an element in the hash table has an address of a JSC-heap object (e.g., a structure object uses a hash table when a new property is added, where a property name is a key and the new structure address is a value). We first create a hash table with the saved hash elements and perform relocation for the structure address.

When the restoration for all objects completes, we relocate those unrelocated addresses that we noted during restoration.

4.5.3 Snapshot for V8

V8 provides a serializer, a utility to save the JavaScript objects [11]. V8 serializer is used to save the built-in objects created during the initialization of the V8 engine in advance. The saved built-in objects will be loaded directly to the JavaScript heap, which can reduce the V8 engine initialization time. We accelerate app loading, not just engine initialization, so we enhanced the V8 serializer to save those

objects created during app loading as well as those built-in objects. We also need to make the serializer to save the DOM objects and events to the snapshot for correct restoration. We first describe how V8 represent the objects in its heap, compared to JSC.

4.5.3.1 V8 Engine Heaps

Figure 30 shows the object representation for the V8 heaps, corresponding to the JSC heaps in Figure 24. As in JSC, there are V8 heap and native heap, which is managed by GC and reference counter, respectively as in JSC. Unlike JSC, however, all objects are created in the V8-heap, except for the DOM elements in the native heap of the browser (and some strings created by the browser); for example, characters of a String object or JIT-compiled machine code exist in the V8-heap, while they are in the native-heap in JSC. Having only the V8-heap objects, implemented more regularly than in JSC, makes the saving and restoring somewhat simpler. Also, unlike the JSC snapshot where objects are saved one-by-one, the V8 serializer can save split objects as shown in Figure 30, which is simpler, yet

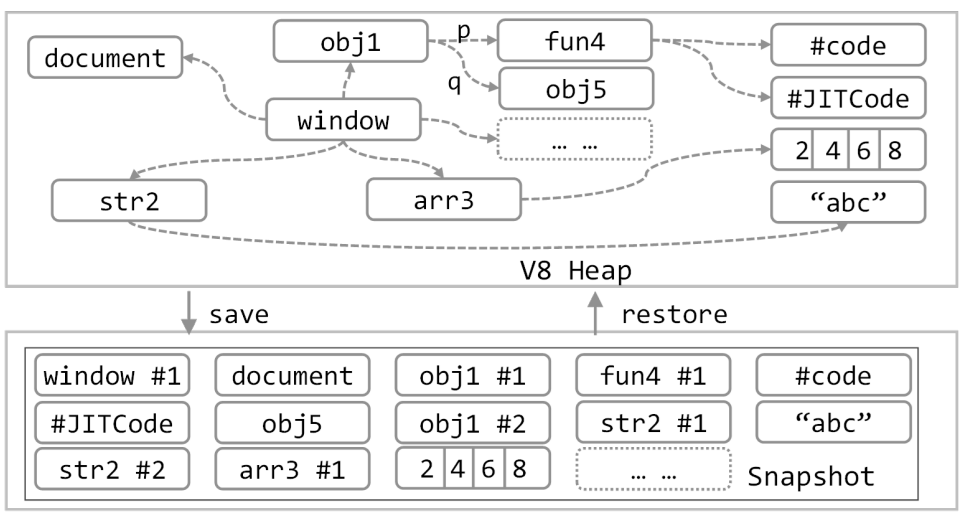


Figure 30. Objects in the V8 heaps and snapshot

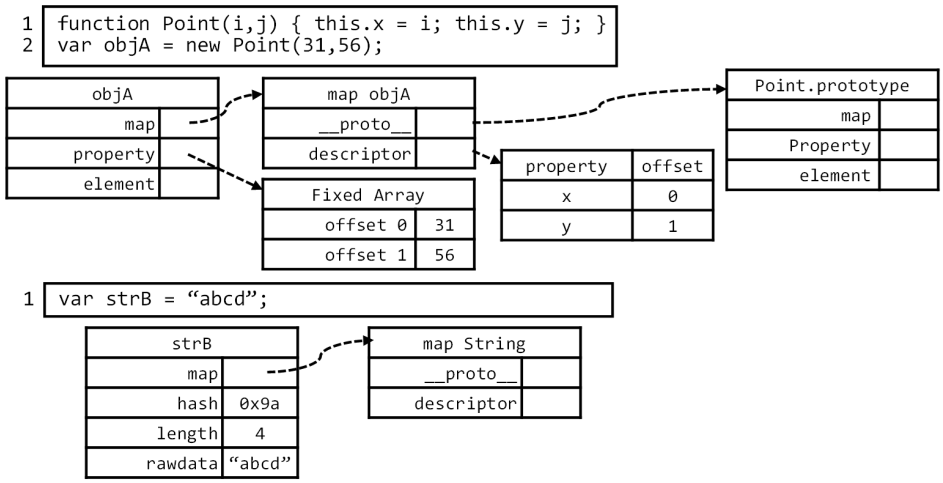


Figure 31. Example of Object and String objects in V8

disadvantageous for advanced snapshot optimization. We will discussed these issues below.

4.5.3.2 Objects in the Heap in V8

Figure 31 shows how the Object and String objects in Figure 25 are implemented in V8. As in JSC, V8 provides a hidden class to accelerate the property access of an object, called a map, which is similarly implemented to the JSC's structure. There are two fundamental differences between V8 and JSC object implementation, though.

The first difference is the way of implementing the C++ classes and the type information for each object. JSC includes many C++ classes, more than the number of basic object types in JavaScript, and they are identified by ClassInfo. V8 has a far fewer C++ classes, and many objects are implemented by the generic Object class. This would simplify the implementation of saving/restoring of objects since there are fewer classes to update. On the other hand, V8 identifies the type of an object using the map field, but since the DOM objects are implemented by generic Objects, it is not easy to

differentiate DOM objects using the map from the generic Objects, making it difficult to save the DOM objects (whereas JSC has a separate C++ class even for each DOM element).

Another difference is how values are saved in the memory of an object. JavaScript employs a tag-encoded value called JSValue to support dynamic typing. For example, V8 encodes the last bit to 1 to represent an integer value. JSC represents only those values in the storage as JSValue, while V8 represents many values with JSValue. This simplifies saving/restoring an object in V8. Serialization requires identifying the object pointers during the saving for the traversal of objects, and V8' s encoded JSValue allows easier identification of pointers, thus simplifying the saving logic for many object types. On the other hand, we cannot know what each field of a JSC object is, so we need to consult the C++ class using ClassInfo, where a separate saving routine will save the individual objects.

Figure 31 also shows that the character data of a String object exist in the V8-heap, not in the native-heap, unlike JSC. This is also helpful for serialization since the objects in the JavaScript heap are somewhat similar, while those objects in the native heap are quite different depending on its kind, so saving each type of a native-heap object (e.g., character data) requires an elaborate saving routine,

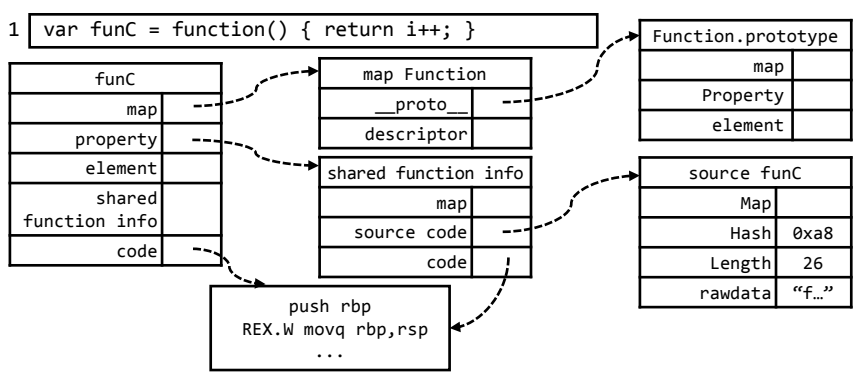


Figure 32. Example of Function in V8

involved with some complexity (see Section 4.5.2.4). Also, character data in the native-heap can be referenced by elsewhere, so handing the reference counter is also considered.

Figure 32 shows the V8 implementation of the function object. Similarly to JSC, there is a Shared Function Info which includes the source code of the function and JIT-compiled machine code (V8 JIT compiler translates the source code directly to the machine code, hence no bytecode). The character data of the source code is in the V8-heap is a regular String object, thus no special routine to save it in the snapshot.

4.5.3.3 Original Serializer in V8

The original V8 serializer saves the built-in objects created during the V8 engine initialization time to a C++ file (const char snapshot[]={snapshot data};), which is then compiled together with the V8 source code. When the V8 engine initializes, the built-in objects included in the V8 executable are deserialized to the V8-heap directly.

We describe how to save and restore the objects. Figure 33 shows how the String object in Figure 31 is saved in the snapshot. V8 object does not include an identifier as the ClassInfo of JSC, which can take care of how to save the objects in detail in the C++ class code. Instead, we first need to access the map of the object to identify the object type and size information. The V8 heap is managed by a unit called space, depending on the object type. So, the space number is saved first, followed by the object size. The V8 map includes some information on how the object is structured, hence how to be saved. There are three cases of saving.

The first case is that saving is performed by a generic saver

routine, which reads a property of the object and if it is a pointer JS Value, we follow the pointer to save the pointed object. An integer JS value is saved as it is. For the String object, the map is a pointer JS value, so the map object is traversed and saved.

The second case is that a sequence of properties does not include a pointer, thus being saved as the raw data. For example, we save all properties of the String object other than the map as raw data.

The final case is that some properties of an object should be saved by a specialized routine. For example, the code of a Function object points the machine code, but it should be saved as an object composed of a prolog followed by the machine code. This cannot be handled by the generic routine, but by a specialized routine (so a Function object is saved partly by the specialized routine (code) and by the generic routine (all other properties).

The snapshot is composed of as in Figure 33. After identifying how to save the properties of a given object, they are listed in order in the snapshot. The first field before adding some item in the snapshot, we add the flag field which informs what is saved in the next slot of the snapshot. For the String object in Figure 33, we first save the flag for the map. Since the map of the built-in types is managed separately by the V8 heap (called root array), so we add

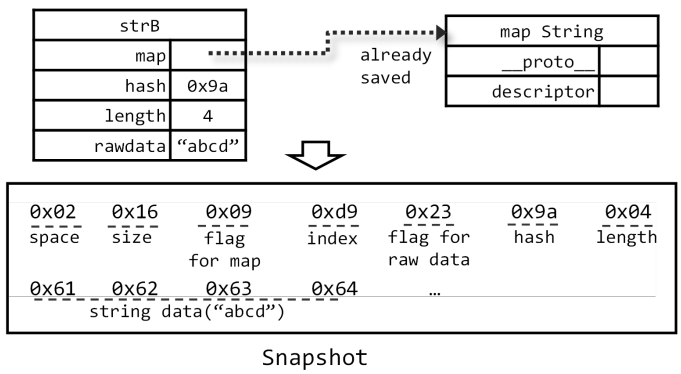


Figure 33. Object save example in V8 serializer

the index of the map. The remaining data can all be saved as rawdata, preceded by the flag for them.

In JSC, when we meet a pointer during the saving, we just mark it but finish the saving for the current object. In V8, however, we access the new object following that pointer and starting to save the object. The remaining object parts will be continued to be saved after returning back. This is the reason what an object can be saved in the snapshot after split, as shown in Figure 30.

Restoration simply reads flags and raw data one-by-one and restores the objects in the V8 heap. Unlike JSC where object-type-specific restore function performs the restoration details, a generic restoration routine will do the job based on the flags.

4.5.3.4 Our Enhancement to the V8 Serializer

In addition to the built-in objects, our loading-time acceleration requires saving those objects created during app loading. Also, we need to save the DOM objects as well as the event objects. So we enhanced the V8 serializer to handle these issues.

We first need to make the serializer to save and restore a great deal more objects. As mentioned previously, V8 heap is composed of spaces depending on the object types. If a space is larger than a single page (a memory management unit of V8) due to many objects, the serializer could not save them correctly. So we incorporated a list of multiple pages during serialization and deserialization to save and restore objects correctly. (considered in newer revision)

When the browser meets a script tag and executes the JavaScript code of the tag, the browser makes a string for the JavaScript source code and delivers to the V8. This string exists in the native heap, thus not savable by the serializer. We made the V8 to relocate this

string to the V8 heap so as to be saved correctly. There is no such issue in the original serializer since it deals only with the built-in objects, not those generated by the execution of a script tag.

Handling of DOM objects and event objects is a more elaborate job, which was not dealt with in the original V8 serializer. We will describe it separately below.

4.5.4 DOM Objects and Event Objects

There are some issues in saving and restoring DOM objects and event objects for both JSC and V8, which will be discussed in this subsection.

4.5.4.1 DOM Objects and DOM Event Handlers

A *DOM object* is a special JavaScript object to interact with the DOM tree. It is a *wrapper* object that points a DOM tree element (node). The *document* object is a built-in DOM object that points the DOM tree root node. Figure 34 shows how a DOM object is created by a JavaScript statement. The DOM API *getElementById()* creates a DOM object, *domD*, in the JSC-heap, whose node property points a DOM tree element, "*gamefield*", existing in the native heap. Since *domD* now points a DOM element, we can register an event handler with a statement, *as domC.onclick = function() {…};*. This will register an *onclick* event handler for the DOM element (not for the DOM object), so if the DOM element is clicked, the registered event handler is invoked.

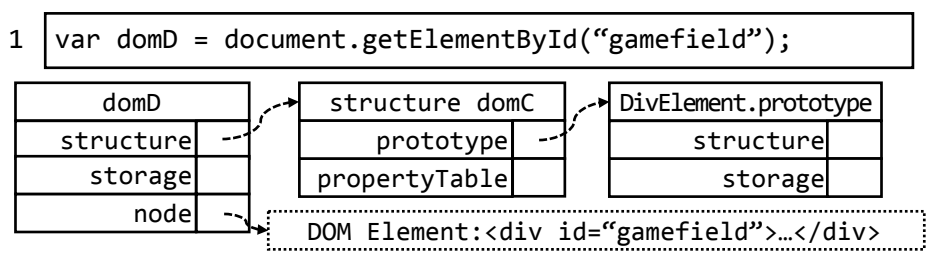


Figure 34. DOM object and DOM element in JSC

It should be noted that a script tag creating a DOM object or registering an event handler for a DOM element is regarded as not updating the DOM tree (see Section 4.4), thus eligible for the snapshot. The problem is how to save and restore these DOM objects and event handlers.

A DOM object created using a DOM API will be accessible when we traverse the objects from the window object. For the JSC case, there is a separate C++ class for each DOM object depending on the DOM element type, which is available in the ClassInfo of the DOM object. So we can tell easily that it is a DOM object and save using the saving routine we provided in its C++ class. For the V8 case, it is hard to tell if an object is a DOM object or a generic object since they have the same structure. When we meet an object during the traversal, first we check object layout in the map to determine whether it has DOM object layout. If it does, we check the last field where a DOM object has the DOM element address, but a generic object uses it for other purposes, then we cast this address to DOM element. Because DOM element has its DOM object address, it can be compared to the object being traversed to certificate that this object is DOM object. In this way, we can identify all DOM objects and save them.

After we identify the DOM objects during the traversal, one issue is how to save the DOM element, the native-heap object pointed by the node property of the DOM object, since it will differ from the current one when we restore from the snapshot; a new DOM tree will be built at the restoration time although it is identical to the one at the saving time. Instead of saving the DOM element, we will save the index number of the DOM element when the DOM tree is traversed in pre-order. So, we will traverse the DOM tree to get the pre-order

index number for each element and save it. For the event handlers, we access each DOM element and save the attached event handler function object, if any, and save the function address and the event type in the snapshot (for the corresponding pre-order index of the DOM element).

When we restore from the snapshot, we create the DOM object with its *node* property set by the current DOM element using the index number. We also register the restored event handler for the DOM element. Finally, the DOM object and element pair is added to the DOM cache.

A JavaScript framework rarely creates a DOM object during its initialization. For example, when *enyo.js* executes for initialization, it accesses only the built-in *document* DOM object for the DOM tree. For efficient DOM event handling, *Enyo* registers its own, event handlers that can handle all events to the *document* object. When any DOM event occurs, these handlers will be invoked first, which will call programmer's event handlers. So we save only the *document* object to the snapshot, and it will point the root of the DOM tree when restored with *Enyo*'s event handlers.

4.5.4.2 Timer Event and Event Handler

Timer events and timer event handlers need to be saved in the snapshot. Timer events are registered using *setInterval()* or *setTimeout()* with the event handler and the time information as arguments. These timer events are also accessible from the *window* object, so we access it to read the pending timer events, and save the event type, event handler, and the time information to the snapshot. When we restore, we simply re-register the timer events

using *setInterval()* or *setTimeout()*. The initialization of the *Enyo* framework does not create a timer event, though.

4.6. Determinism for Snapshot

Our Snapshot technique creates snapshot before app loading, and uses it to accelerate app loading at actual app execution. Therefore, snapshot saving point and restoration point is different from each other. When snapshot is restored, app state at snapshot saving time is restored as it is. The problem is that this may cause unwanted situation in some apps. For example, a simple clock app gets current time by *Date* function during app loading. If snapshot is saved for this app, this snapshot contains state of saving time. When app is loaded later using this snapshot, it will show time saved in snapshot, rather than current time which actually expected to show. In this case, snapshot is not working correctly. In other words, for snapshot to work correctly, loaded state of the app should always be fixed. We call this case that app has deterministic execution state. As this study saves JavaScript state of app loading time, we discuss about determinism for JavaScript.

The most basic situation of nondeterminism is modification of source code. If source code is modified because of some reason such as app update, its loading state is also changed. In this case, snapshot needs to be created again. However, this kind of snapshot re-creation is not frequent because app update rarely occurs relative to app execution.

JavaScript execution state is determined by the source code and input. Although same source code is used, different input may modify app execution state. Some of these inputs are created by users, but others appear regardless of user actions. This can be seen as

nondeterminism of JavaScript. If this nondeterminism appear before snapshot saving time at app loading, snapshot should be applied before nondeterminism occur or snapshot cannot be applied properly.

One of the nondeterminism occur during JavaScript execution is random value created by call of Math.random() and value from functions related to Date. These values are changed every time the app is executed. As we cannot expect execution stated related to these values. Snapshot cannot be applied if this case appears before snapshot saving time.

Another nondeterminism is about storage and browser. Web app can save data in client device using cookie or localStorage. If data in the device at restoration time is different from one at saving time, snapshot cannot be applied. Also, information about browser or state of browser window can be obtained through navigator or screen. However, as browser or device is same when creating and restoring snapshot, nondeterminism does not occur in this case. In other words, navigator or screen returns same value when saving and restoring snapshot.

Lastly, order of event handler call is also nondeterminism. While app loading, resources like image and multimedia object are loaded and event handler can be called when each resource loading is completed. In this case, order which resource loading is completed may differ for each app loading.

As we mentioned above, there are many nondeterminism issues while web app loading time. In this study, we did not propose way to solve this issues. Rather, we recommend developers or users to set snapshot saving time before nondeterminism occurs. App loading time acceleration becomes less effective as snapshot saving time becomes earlier. However, as we showed in experimental results,

JavaScript execution time is optimized enough with snapshot. This is because many web apps are written using JavaScript framework. Like enyo app explained in Figure 22, JavaScript framework provides not only APIs but also development model for app. Until line 6, 7 and *new HelloWidget()* of line 10 in Figure 22 (a) snapshot can be applied for most cases. For frameworks other than enyo, nondeterminism does not occur when executing framework itself and app initialization. Here, function call related to storage or browser does not occur at all and event handler is registered while executing this code, which means event handler related to resource loading is not yet fired. Problem is, date or random function may be called. However, nondeterminism caused by date or random at this time may not affect actual app loading.

Figure 35 shows code about *Date* function called while enyo.js execution. Value of timestamp property is decided by value of Date. However, this code is used as timestamp for maintaining cache, which means that actually wanted value can be obtained regardless of the value of timestamp. As *fileCache* is not used while enyo.js execution, timestamp value is also remain unused. When function which reads file occurs after enyo.js execution, timestamp value is then changed to actual Date value. Thus, Date here in enyo.js does not affect app loading using snapshot. For many other frameworks, Date and Math.random is used to assign unique identifier to object property.

```
1  //in getJsonFile :
2  //  load file from disk and convert to Json
3  if (_fileCache[cachePath] !== undefined) {
4   json = _fileCache[cachePath].json;
5  } else {
6   _fileCache[cachePath] = { path: cachePath,
7    json: json, locale: params.locale,
8    timestamp: new Date()};
9  }
```

Figure 35. Example of using Date in enyo.js

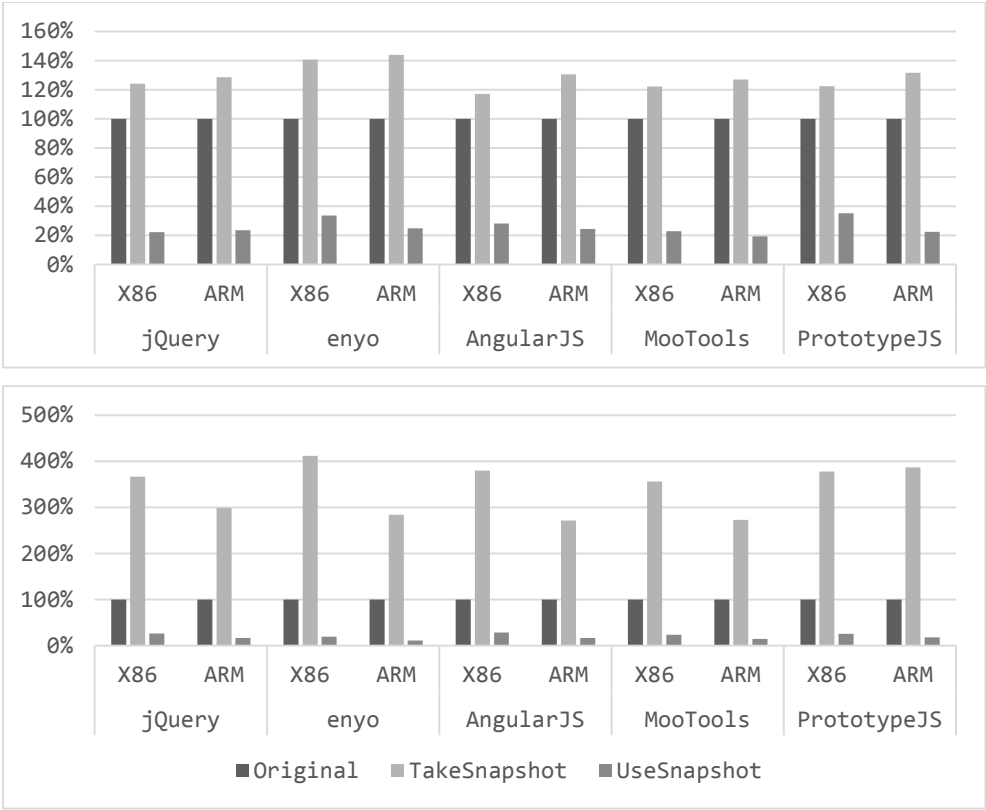


Figure 36. Framework initialization time.

These cases are nondeterminism in strict sense, but does not affect actual behaviour of apps.

In many web apps, nondeterminism may bring snapshot saving time forward regardless of DOM modification. However, most of framework and app initialization codes have deterministic execution state, and thus snapshot can be applied properly. When developing web app, app developer may put nondeterministic part at the latter part of app loading, which can make snapshot optimization more effective.

4.7. Experimental Results

This section evaluates performance gain on loading time.

4.7.1 Acceleration of Framework Initialization

We first evaluated the snapshot for the web framework on a Pandaboard ES [35](ARM Coretex-A9 1.2GHz, 1GB RAM) using a WebKit browser rev-149728 for JSC and Blink browser 36.0.1943.1 for V8. We ran the app in Table 3. We measured the running time 10 times and took an average.

Figure 36 shows the initialization time of the five frameworks for both JavaScript engines on x86 and ARM. It compares the original initialization time (100%), the initialization time added with the overhead of taking a snapshot, and the initialization time accelerated by the snapshot. The framework initialization time is reduced consistently for the three frameworks by 82% for ARM and 77% for x86, on average. The overhead of saving a snapshot is around 29~312%, yet it would occur only once in most cases, so it is not an issue.

We also measured the snapshot file size of each framework in Table 4, which is around four times larger than the original file size of its JavaScript source code.

Table 5 shows the distribution of objects saved in the snapshot file for each framework in JSC and V8. The first column in the JSC table shows the number of objects allocated in the JSC-heap, and the second column shows the number of JavaScript objects among them,

Table 4. Snapshot Size (ARM)

	Source Code	Snapshot in JSC	Snapshot in V8
jQuery	94 KB	360 KB	1,547 KB
Enyo	598 KB	2,389 KB	3,245 KB
Angular JS	123 KB	351 KB	1,416 KB
MooTool	152 KB	563 KB	1,733 KB
PrototypeJS	194 KB	543 KB	1,720 KB

Table 5. Distribution of Objects Saved in the Snapshot.

JSC

	JSC-heap Objects	JavaScript Objects	Function	String	Object	Array	RegExp	DOM	Native-heap Objects
jQuery	3060	1375	604	417	100	37	169	5	1646
Enyo	18901	9038	2982	2820	2598	591	11	1	8972
AngularJS	2804	1282	576	477	54	75	63	2	1601
MooTools	5202	2148	1262	665	116	42	13	3	2042
PrototypeJS	4067	2683	782	674	91	35	47	4	2138

V8

	V8 heap Objects	JavaScript Objects	Function	Object	Array	RegExp	DOM	String
jQuery	27922	4261	2297	1766	415	153	5	5144
Enyo	59092	9557	4736	4769	966	16	1	14924
AngularJS	25167	4071	1236	2265	478	65	2	4103
MooTools	32052	4779	1365	2953	413	21	3	5560
PrototypeJS	29230	4287	1352	2474	396	37	4	5357

which is around half of the JSC-objects; the remaining is for the helper objects such as structures, scopes, etc. The distribution of the JavaScript objects is also depicted, where function objects and string objects are dominant. The JavaScript framework is mostly composed of library API functions, so there are many function objects created during the framework initialization. When a function object is created with a name (e.g., *function foo() {…}*), it can have a property with a name string, leading to the creation of many String objects as well. There are a few DOM objects since DOM objects are not created much in the framework. The final column in JSC shows the number of native-heap objects.

The V8 table shows a similar distribution. There are no native-heap objects in V8 since every object exists in the V8 heap. V8 classifies the String object, identifier, the function code as the string objects (and we cannot distinguish each type), so we separated them from JavaScript objects and depicted as String in the table.

Table 6. Enyo apps (http://enyojs.com/showcase/).

A	Enyo2 Sampler (Enyo/library showcase)
B	Hello World
C	CryptoTweets (A simple puzzle game)
D	Bing Maps (Enyo-based map)
E	PiratePig (An HTML5 canvas app)

4.7.2 Acceleration of Enyo Apps

We also experimented with the five Enyo apps in Table 6 to see if the snapshot is working and useful for real apps. This experiment is also performed on the same Pandaboard ES.

All Enyo apps have a HTML file structure similar to Figure 37. It invokes *enyo.js* and *app.js*, followed by the creation of the *App* object by *new App()*, none of which affects the DOM tree. Then, it calls the *renderInto()* function of the object, which modifies the DOM tree. So, we experiment with two cases of the snapshot: a snapshot for *enyo.js*+*app.js* and a snapshot for *enyo.js*+*app.js*+*new App()*. For

index.html

```
1  <!DOCTYPE html>
2  <html>
3   <head>
4    <title>App Title</title>
5    <link href="enyo.css" rel="stylesheet"/>
6    <link href="app.css" rel="stylesheet"/>
7    <script src="enyo.js"></script>
8    <script src="app.js"></script>
9   </head>
10  <body class="enyo-unselectable">
11   <script>
12    new App().renderInto(document.body);
13   </script>
14  </body>
15 </html>
```

Figure 37. HTML file for Enyo apps

the latter case, we made *new App()* as a separate script tag and made a snapshot right after it.

One big difference of the Enyo app from the jQuery app is that the Enyo framework is modularized so that only those framework modules used by an app can be included in the app. That is, when the programmer develops an app with Enyo kinds and runs the deployment, a customized *enyo.js* is generated. This *enyo.js* is smaller than the whole *enyo.js* framework in Table 3 and is different for each app in Table 6. The *app.js* includes the Enyo kind code but it can also include some framework code as well. Despite these variances, we could obtain consistent results, as below.

For each app, Figure 38 (a), (b), (c) show the app loading time for the original, for the snapshot of *enyo.js+app.js*, and for the snapshot of *enyo.js+app.js+new App()*, respectively. The loading time is measured from the start until the *onload* event fires. It is divided into three parts: *enyo.js+app.js* time, *new App()* time, and *others*; *others* include the HTML parsing, CSS styling, layout and rendering for the DOM tree, and a call of *renderInto()*. On average,

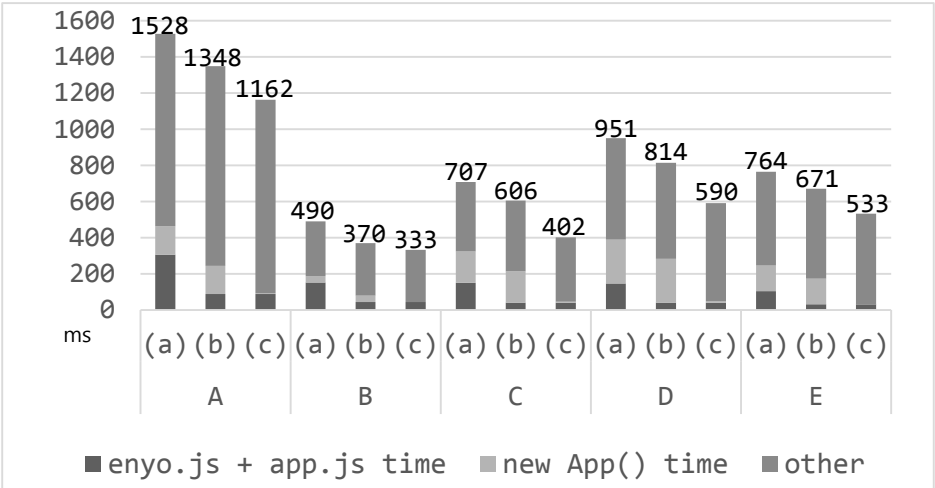


Figure 38. The app loading time (a) original, (b) *enyo.js +app.js* snapshot, (c) *enyo.js+app.js+new App()* snapshot

Figure 38 shows (b) and (c) reduce the original loading time by 15% and 33%, respectively. This is due to the reduction of *enyo.js+app.js* time by 71% in (b), slightly lower than in Figure 36. What is interesting is that *new App()* time is reduced by almost 98% in (c). We found that *new App()* creates only 10% of objects created in *enyo.js+app.js*, but makes many function calls to inherit Enyo kinds, so its execution time is comparative to *enyo.js+app.js*. Skipping *new App()* with the snapshot, thus, has a much better impact due to lower restoration overhead.

Chapter 5. Enhanced Snapshot Optimization

5.1. Limitation of JavaScript Snapshot

Snapshot Optimization is proposed to improve loading time of Web app. [12] Snapshot Optimization saves JavaScript execution state (Execution Context) which is repetitively executed while app loading and recovers snapshot to start app loading from saved state. There is much JavaScript execution while Web app loading, thus Snapshot Optimization which omits JavaScript execution can remarkably reduce Web app loading time. However, Snapshot Optimization has many restrictions to apply on Web app.

We propose two idea which can further improve Snapshot Optimization.

Existing Snapshot Optimization only considered JavaScript execution state, and it cannot optimize full repetitive JavaScript execution at loading time. This is because DOM-related operation occurs while JavaScript execution is not considered. If DOM manipulation occurred at loading time can be saved, it can reduce loading time of web app more than Snapshot Optimization.

Another problem of Snapshot Optimization is the matter of size. In order to fit ECMAScript specification and optimize dynamic characteristics of JavaScript, JavaScript engine creates not only actual JavaScript objects but also many additional objects like Hidden Class or Prototype. Because of this, there are large size overhead compared to the source code. Snapshot size overhead is at most 10MB, which is 10 times larger than source code. This size overhead does not matter in general desktop environment. Recent mobile devices have large memory thanks to the advance of memory

technology, but flash memory which is mainly used as storage in mobile device has limited lifetime of file writing so small file size always matters. Also, in case of IoT device even flash memory has small capacity, which leads to considering file size.

Issue about file size can be resolved by using file compression. However, file is decompressed at actual Web app loading time and this may slows loading time. We perform file decompression and snapshot recovery in another thread to hide overhead by decompression.

Another characteristic of Web app is that it is written with JavaScript Framework made with JavaScript. This means that many apps actually have same objects created from JavaScript Framework. If many apps share objects created from Framework, large parts of each app's Snapshot can be removed. We will call this Framework Snapshot Sharing.

5.2. Architecture for Enhanced Snapshot Optimization

Figure 39 shows the structure using our Enhanced Snapshot to improve Web App loading performance and minimize size of the

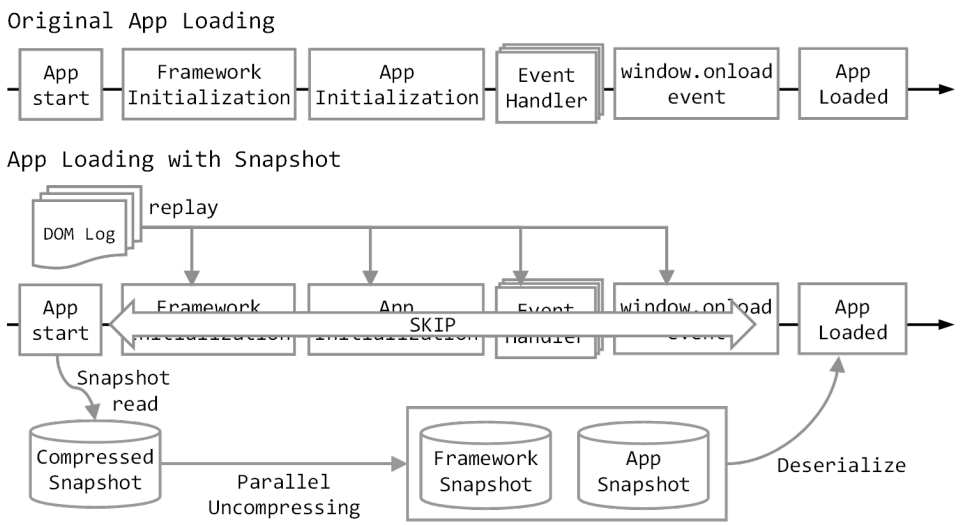


Figure 39. Architecture for Enhanced Snapshot Optimization

Snapshot. Existing Web App go through a series of JavaScript execution after App is started. First, it loads Framework by executing Framework JavaScript code. Then App code is executed and event handlers which are registered by previously executed JavaScript is called. These event handlers mainly correspond to load event of DOM element of resources like images. After parsing of HTML file is over, window.onload event is finally fired and App loading is completed. Loaded app works in event-driven way by previously registered event handlers. When Snapshot is applied, decompressing thread performs decompression on Snapshot when App is first started. Framework Snapshot and App snapshot are separately compressed, so Framework is decompressed first and then App snapshot is decompressed. App performs parsing of HTML file like before to construct DOM tree. If script is met at this time, execution of script is skipped. Instead, DOM log which is created from this script is replayed to recover DOM state. Finally, when window.onload is fired, (here, event is fired but there is no JavaScript execution, because there is no event handler.) Decompressed Framework Snapshot and App Snapshot is read together to recover JavaScript state. In this way, App loading can be accelerated.

Figure 40 shows HTML code of jQuery Web App, which uses jQuery generally used to develop Web app. Also, it shows how to use Enhanced Snapshot Optimization implemented in this research.

Until which point the app loading sate is always same while app loading time is decided by how App developer writes app. If whole loading time of Web app is same, that is, Web app always go through same operation until *window.onload* event handler is called, snapshot attribute is added as attribute of title element like Figure 40 (a) to apply Snapshot. Or, it is possible to save state until execution

```
<!doctype html>
<html>
<head>
 <title snapshot=true>App</title>·····(a)
 <script src="jQuery.js", snapshot=true></script>
 <script src="app.js",snapshot=true></script>···(b)
</head>
<body>
  <div> … </div>
</body>
</html>
```

```
var width = 500;
…
$(document).ready(function() {
          /* has DOM change code */ });
//ready : jQuery API for DOMContentLoaded Event
```

Figure 40. Snapshot attribute for Enhanced Snapshot

of specific script as snapshot like shown in Figure 40 (b). All loading time operation of app in Figure 40 is always same means that execution of JavaScript is always fixed and DOM APIs executed through JavaScript state, or DOM state at loading time, is always same. Web app loading includes script evaluation executed through script element together with call of event handler registered with *DOMContentLoaded* or *onload* event. If this execution of event handlers is totally same, we can skip all operation and correctly load Web App through state recovery from snapshot, without actually executing any JavaScript code.

5.3. DOM Log-Replay

This section explains DOM Log-Replay Approach which can save or restore DOM state..

5.3.1 Why DOM Log-Replay?

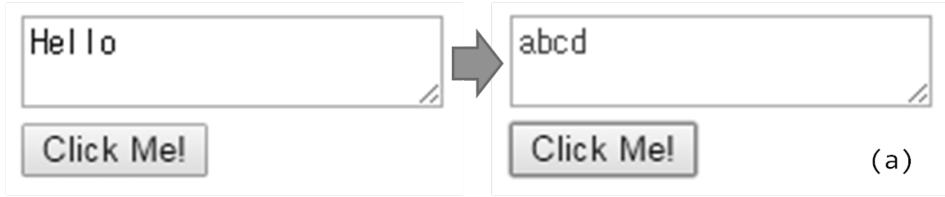
DOM state can be saved by diverse approach. First, like we saved JavaScript state, we can visit each DOM node can dump its memory.

However, this approach requires very complex implementation overhead. Each DOM node has different structure according to its type, and there are very complex nodes like ones related with multimedia. So, saving each DOM node will be hard to implement and debug.

Another approach is to save DOM tree state as string format. By using outerHTML API which converts DOM node to string, current DOM tree can be simply converted to string. Figure 41 shows an example using outerHTML. In Figure 41 (c), color of the text shown in screen is changed to red by line 5. Then, when button is clicked, event is fired and the text is changed to "abcd". This means that there are two DOM API calls.

As being seen in line 4, outerHTML makes string which contains current DOM element and all child nodes and all attribute information is also included in the string, so it is easy to represent current DOM tree. Modification of specific attribute is reflected, as in line 8. However, outerHTML cannot reflect all attribute changes through DOM API call. In line 13, value of the element is changed to "abcd" but this change does not applied to the string, resulting in same outerHTML in line 8. This means that saving DOM state with outerHTML is not perfect. Especially for media objects like canvas and video, it is hard to represent image of canvas or information of playing media with string. Also, there are elements like TextFragment which cannot be represented with outerHTML.

Compared to these approaches, DOM Log-Replay is simple and can save all DOM state unlike outerHTML. After HTML file is parsed by browser and DOM tree is created, all changes in DOM is performed via execution of JavaScript DOM API. Saving log about



(a)

index.html

```
1  <!doctype html>
2  <html>
3  <head> <title>Demo</title> </head>
4  <body>
5   <textarea id="text">Hello</textarea>
6   <button type="button" id="btn"> Click Me!</button>
7   <script src="app.js"></script>
8  <body></html>
```

(b)

app.js

```
1  var text = document.getElementById("text");
2  var firstHTML = document.body.outerHTML;
3  alert(firstHTML);
4  //"<body><textarea id="text">Hello</textarea><butt
   on type="button" id="btn">Click Me!</button><scrip
   t src="app.js"></script></body>"
5  text.style.color="red";
6  var secondHTML = document.body.outerHTML;
7  alert(second);
8  //"<body><textarea id="text" style="color: red;">
   Hello</textarea><button type="button" id="btn">
   Click Me!</button><script src="app.js"></script>
   </body>"
9  document.getElementById("btn").click =
10 function() { text.value="abcd"
11  var thirdHTML = document.body.outerHTML;
12 alert(thirdHTML); }
13 //"<body><textarea id="text" style="color: red;">
   Hello</textarea><button type="button" id="btn">
   Click Me!</button><script src="app.js"></script>
   </body>"
```

(c)

Figure 41. DOM outerHTML example

DOM API called in JavaScript and replaying the log to restore DOM state can be simply implemented, not needing complex debugging.

Media-related DOM objects such as Canvas and Video, which is supported in HTML5, are also manipulated through API called in JavaScript. Thus, DOM Log-Replay can reproduce operations on these Media objects. Also, there are device APIs like

Navigator.vibrate which makes the device vibrate is added with HTML5 support. Currently our implementation only supports log about DOM manipulation but as device API works same with API calls in JavaScript, Log-Replay of device API can be implemented.

5.3.2 Capturing DOM Log

DOM API logging for saving DOM state is very simple. All DOM API function calls occur during JavaScript execution are recorded with their arguments, which can be strings or DOM nodes. There are DOM APIs which are called only to get information of DOM element for later usage in JavaScript such as *getAttribute*. These functions does not change DOM state, and we ignore these functions and only makes logs about APIs which actually changes DOM state.

Figure 42 shows format of the log for DOM API. API id stands for the identifier of API which is manually given by us. All DOM API uses integer, string, element/node as type of their arguments. String has its own implementation in browser but there are some unnecessary information in it. Thus, we only save length and characters of the string and then create new string with the length and characters when replaying. Main issue occurs here is how to record node/element in the log. Naïve approach is to record the index of corresponding node/element in the DOM tree. However, not all DOM node exist in DOM tree. For example, a DOM element created by

Attribute change (setAttribute)

API id	Target Element (element)	Attr Name (string)	Value (string)

DOM node append (appendChild)

API id	Target Node (node)	Attached Node (node)

Figure 42. DOM API log example

document.createElement API is not in DOM tree unless it is attached to the DOM tree. Thus, we need to find the way which can index these cases. We added a node array which is managed internally to solve this problem. This array saves not only the document element which is the root of DOM tree, but also other roots of tree consists of DOM elements. For example, when *createElement* is called and an attribute is added to this created element, we produce a log which append created node to the node array after calling *createElement*. We call this *RegisterNode*. Then, at the point which replays attribute adding log, the element is saved in node array and can the index can be used. Index is consists of a pair of two number, array index and tree index. Array index stands for the node array index of the root for corresponding element, and tree index stands for the position of corresponding element in the tree. Document element is always the root of the basic DOM tree, so 0^{th} index of node array is always document element.

Additionally, as JavaScript execution occurs while parsing the DOM tree, we added divide log to distinguish which log is created in which script. Divide log is inserted in the DOM log when new script element is being parsed.

5.3.3 Replay DOM

While Web app loading, DOM log is read and replayed if DOM log exists. Replay is to read log and call DOM API which corresponds with the log. We added divide log at log saving time, so we can find which script was the log executed. If script element is met while DOM tree parsing, browser replays DOM log of the script instead of executing the script itself.

Index is saved as {array index, tree index} in log, so we can find DOM node with this information. First, we read array index to find root saved in Node Array, and then find DOM node in the tree which corresponds with the tree index. In case of string, we create string object with saved character data and length of the string so that DOM API can be correctly called.

5.4. Pre-Decompressing Snapshot

The simplest way to reduce size overhead of snapshot tis to use file compression. Much part of the snapshot contains strings such as JavaScript source code. Addresses in the snapshot also have many redundant values. This makes compressing snapshot more advantageous in size. However if snapshot is compressed, decompression is needed at Web app loading time before deserializing snapshot. In order to fully optimize web app loading time, this decompression overhead must be minimized.

Since multicore environment is now popular, we can hide decompression overhead by performing snapshot decompression and deserialization in another thread.

We used lzo [36] as a file compression library. Lzo divides file into blocks of specific size. Compression and decompress is done for each block in order. Snapshot deserializer also reads snapshot from the front to restore saved state. Thus, deserializer do not need to wait full decompression of the snapshot file. Instead, if a block of the snapshot is decompressed, deserialization of the block can be immediately started. Figure 43 shows naïve approach which waits decompression of the snapshot and then performs deserialization and pre-decompressing approach which decompress the snapshot in advance and performs deserialization immediately. In case of naïve

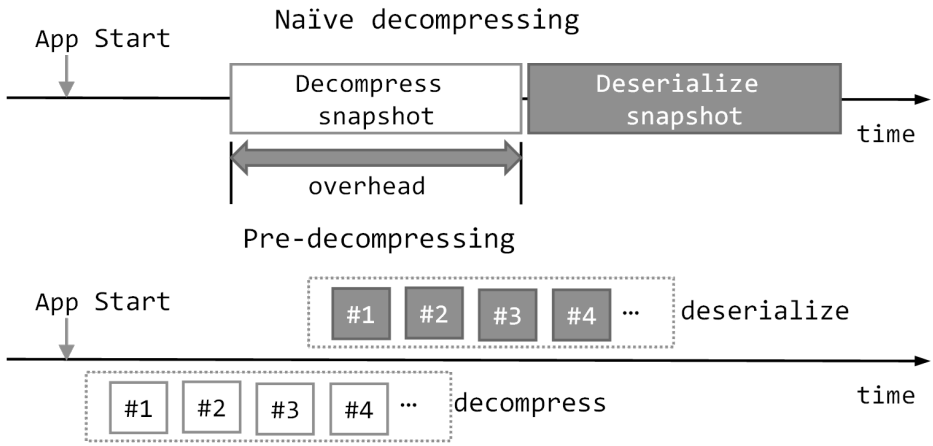


Figure 43. Naïve and Pre-decompressing snapshot

approach, file decompressing time directly becomes overhead to whole loading time. Whereas, in pre-decompressing approach, file decompression overhead is hidden by deserialization which is done ahead of time and parallel with decompression.

To correctly deserialize snapshot, we need to check if whole block containing object which is to be deserialized is decompressed or not. Obj #N of Figure 44 exists along block #K and block #K+1. In this case, deserializer waits until decompression of block #K+1 is completed and then starts deserialization. Each block has a flag and deserialize thread can access this flag to check whether decompression of the block is completed or not. Decompression of the file starts as app loading start rather than the point snapshot is actually needed. So for most cases, decompression done at the time when snapshot deserialization starts and wait does not occur.

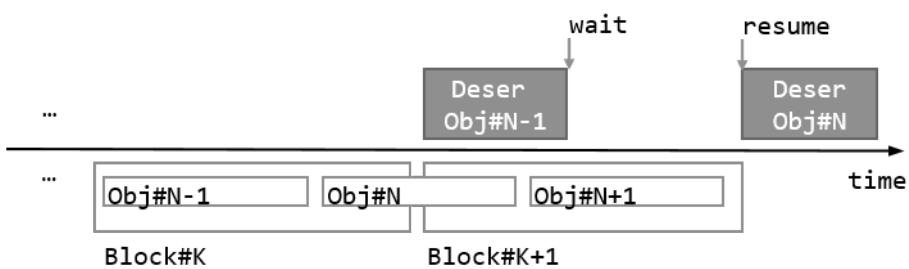


Figure 44. Wait point for Deserializer

5.5. Framework Snapshot Sharing

One characteristic of Web app is that it uses JavaScript web framework to write an app. JavaScript web framework such as jQuery [1] supports various API or app development model to ease app development. JavaScript framework itself is also written in JavaScript and used by embedding source code into HTML file like other JavaScript file. (e.g. *<script src=" jQuery.js" ></script>*) Because app code which developer is written can use API of the framework only when web framework is loaded, script of framework is embedded foremost in the HTML file. Thus JavaScript code of framework is executed first and then app specific code is executed. This means that web apps which use same web framework has same JavaScript state at some point. In other words, snapshot data of web framework redundantly exist in snapshots of each apps. If we can divide framework snapshot which only saves state after web framework and app snapshot which saves app specific objects, framework snapshot can be shared between multiple apps and app loading can be done with shared framework snapshot and app snapshot of each app. This can dramatically reduce size overhead because app snapshot has far less objects compared to normal snapshot.

However, framework objects cannot be simply separated from other objects. Saving snapshot is to save objects exist in JavaScript heap at saving point, so each object cannot be distinguished whether it is framework object or not. Framework objects and other objects are not completely separated. App specific code can modify framework object, or some object can reference framework object. So we have to solve issues on finding framework object which is

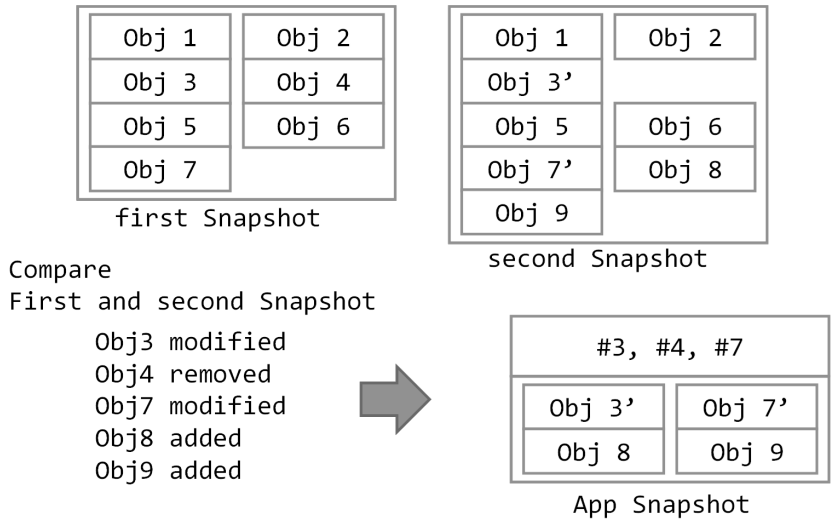


Figure 45. Example app snapshot

modified by app specific code and dealing with references between framework snapshot and app specific snapshot. Especially framework snapshot is used by different apps, and created at different timing with each app snapshot, so reference information to framework object in app specific snapshot and information of framework snapshot is different. For correctly restore state from both snapshot we need to match these information.

We implemented new algorithm for saving app snapshot. Framework snapshot which is shared between web apps is saved after only framework is loaded.

App snapshot is created by making two different snapshot. First, after framework JavaScript is executed, first snapshot is created. For convenience we used framework snapshot instead of actually executing framework code, so first snapshot is taken right after state is restored with framework snapshot. That is, first snapshot is exactly same with framework snapshot.

Second snapshot is created at the point which actual optimization is needed. Unlike comparing with framework snapshot, first snapshot

and second snapshot can be easily compared because they have actually same addresses. We check if objects saved in first snapshot remains same in second snapshot. This process simply compares binary data written in snapshot and let us know which object is changed or removed. When objects in first snapshot are checked, we find newly added objects in second snapshot compared to first snapshot. So we can recognize which object is modified, removed, or added compared to framework snapshot. Finally, with list of removed or modified objects and snapshot data of newly added object, we create app snapshot. Figure 45 shows an example of creating app snapshot by comparing first snapshot and second snapshot.

Next problem is that framework snapshot and app snapshot have different address for same object because they are created separately. Because framework snapshot is shared, we will solve this problem by relocating reference in app snapshot which points to the object in framework snapshot.

Framework snapshot and app snapshot are deserialized together, not separately. First, we read the list saved in app snapshot. This list contains objects modified or removed by app, so while deserializing framework snapshot we skip objects in this list. After deserialization of framework snapshot is completed, app snapshot is deserialized and state restoration is done.

5.6. Evaluation

5.6.1 Environment

We evaluated our enhanced snapshot for the web applications on a Pandaboard ES [35](ARM Coretex-A9 1.2GHz, 1GB RAM) and Odroid-XU4 [37](ARM quad ARM-A15 2.0GHZ & quad ARM

Table 7. Benchmark Web Apps

Enyo Apps (http://enyojs.com/shwocase/)	
A	Bootplate
B	PiratePig (An HTML5 canvas app)
C	CryptoTweets (A simple puzzle game)
D	Enyo 2 Sampler (Enyo and Library showcase app)
jQuery Apps	
E	LightFlip (http://10k.aneventapart.com/2/Uploads/598/)
F	EmotiColor (http://emoticolor.blogspot.kr/)
G	Cubeout (http://alteredqualia.com/cubeout)
H	SimplePlanner (https://github.com/knadh/simpleplanner)

Coretex-A7 1.4GHZ, 2GB RAM). We use WebKit browser rev-149728.

5.6.2 Benchmark Web App

We experimented with Enyo and jQuery app which is widely used as a JavaScript web framework. The benchmark web apps are listed in Table 7.

5.6.3 Acceleration of Web Apps using DOM Log-Replay

Figure 46 shows measured web app loading time using JavaScript snapshot with DOM Log-Replay which is proposed in this paper. When DOM Log-Replay is not applied, performance improvement is average 1.12x / 1.11x. When DOM Log-Replay is applied, result is increased to average 1.87x /1.52x. This result shows that for jQuery app DOM Log-Replay is especially effective. Although more JavaScript execution is removed by applying DOM Log-Replay, deserialization time which restores actual JavaScript state does not increase much. This characteristic shows that most objects are created at framework loading or early part of app specific code, and then DOM API or created functions are called to deal with remaining

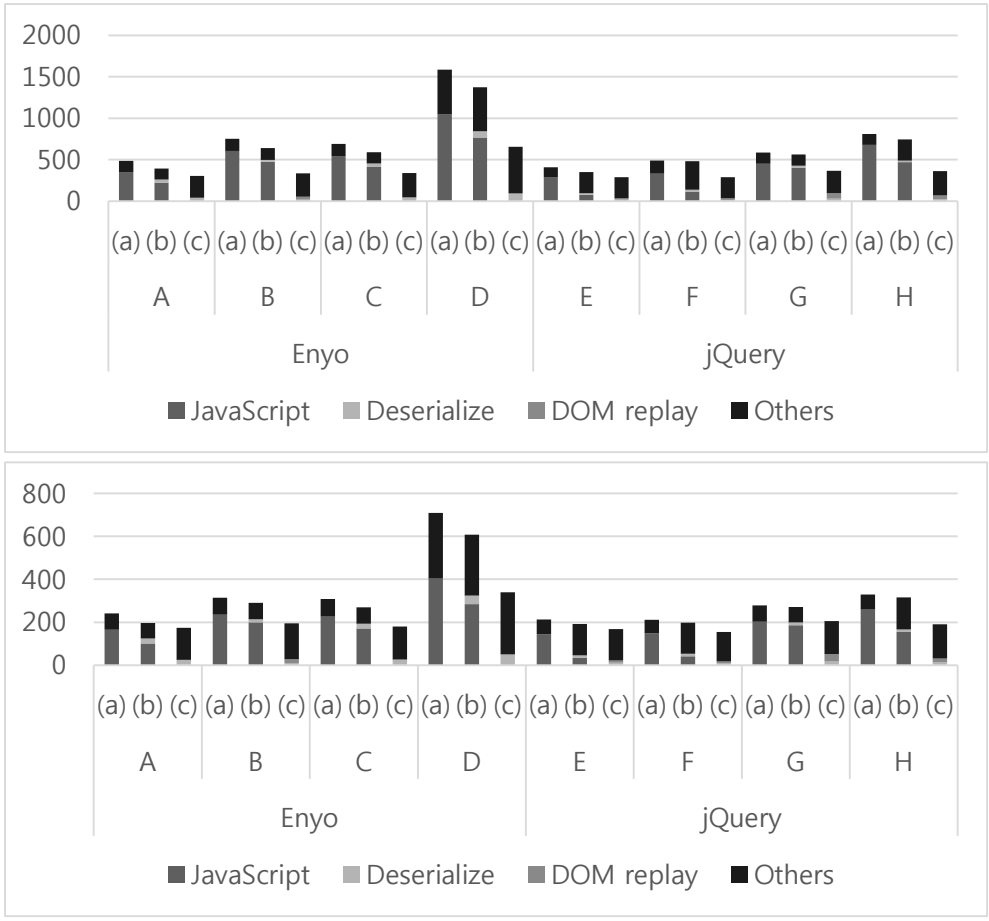


Figure 46. The app loading time (a) original (b) snapshot with no DOM (c) snapshot with DOM Log-Replay
Pandaboard (top), Odroid (bottom)

execution. Another characteristic is that "Others" part becomes very dominant when DOM Log-Replay is applied. This can be explained as following: at original app loading, JavaScript execution or DOM tree parsing is performed parallel with layout and rendering. However, when app is loaded using snapshot with DOM Log-Replay, JavaScript execution is fully removed so the parallel execution part for layout and rendering is included in "Others". We compared "Others" part with the loading time of HTML file which has same DOM tree with DOM tree generated by web app loading, and confirmed that two value is almost same.

Table 8. Compressed Snapshot size (KB)

KB	Source Code	Original Snapshot	Compressed Snapshot
A	267	1240	458
B	152	922	327
C	271	1465	502
D	635	2930	979
E	92	340	139
F	94	343	141
G	160	928	312
H	107	506	192

5.6.4 Memory reduction by Pre-Decompressing

Table 8 shows reduce snapshot size by compression. Because snapshot contains many redundant address data and string data, compression rate is quite good and size is reduce to about 36%. This leads to reduction of overhead from 4.8x to 1.7x.

Figure 47 shows comparison of naïve approach which serially decompresses snapshot before deserialization and pre-decompressing approach which decompresses in advance and deserializes immediately relative to original loading time. Although decompression time does not occupy much portion of whole loading time, result shows that pre-decompress absolutely does not delay

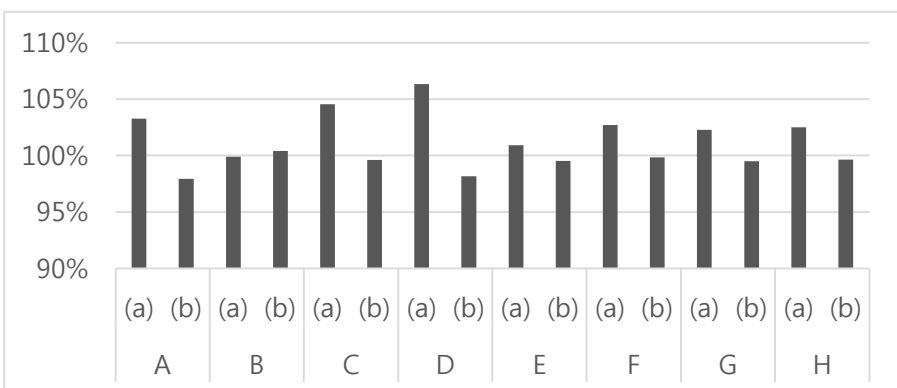


Figure 47. Serial(a) and Pre(b) Decompress performance

loading time. Rather, there was a small speed-up because reading snapshot is done in advance. In case of Pandaboard, a little performance decrease is shown because of the other thread and scheduling used by the web browser, since Pandaboard is a dual core environment.

5.6.5 Memory reduction by Framework Sharing

Table 9 shows measured size of snapshot using Framework Sharing. Because enyo apps do not use same framework file although it uses same version of framework, this evaluation is only done on jQuery apps. All jQuery framework files of jQuery apps are changed to same version (jQuery 1.11.1) for evaluation. When all apps share framework snapshot of jQuery, size of some app snapshot is rather smaller than size of compressed snapshot in Section 5.6.4 In addition, compression can be applied to app snapshot so combining both optimization shows very small size overhead. When four apps used in the evaluation shares same framework file the size overhead turns out to be 12x. If Framework Sharing and snapshot compression is applied the overhead is reduced to 2.59x. Furthermore, web app loading with snapshot does not need JavaScript source code so snapshot optimization is possible with only adding memory of 1.59x of source code size, which is quite small.

Table 9. Framework Sharing (KB)

Framework Snapshot : 392 / 182				
	Original Snapshot	Snapshot _compress	App Snapshot	App Snapshot _compress
E	372	152	52	22
F	375	153	54	22
G	928	312	611	179
H	525	199	213	70

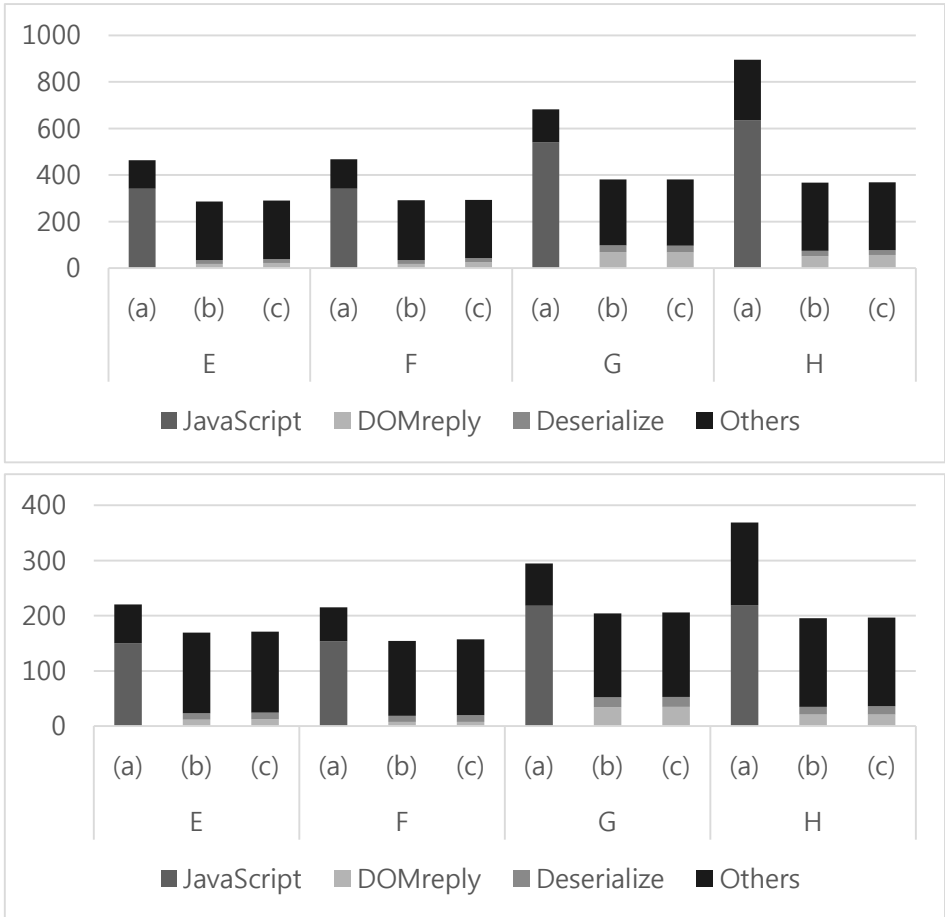


Figure 48. Loading time of Web apps with framework sharing and pre-decompressing: (a)original (b) snapshot (c) snapshot with framework sharing and pre-decompressing, Pandaboard (top), Odroid (bottom)

Figure 48 shows loading time performance of snapshot without memory optimization and snapshot with Framework sharing and Pre-decompressing. Regardless of the board type, there is almost no performance overhead of memory optimization.

Chapter 6. Related Works

Snapshot has originally been introduced for the database [38], but it has also been used to save the state of applications or the virtual machines for diverse purposes.

VMware employs job migration to send an app from the smartphone to the desktop so that the computation is made at the desktop and the result is sent back to the smartphone using the VMware image [19]. However, the VMware is a system VM, which includes the whole Linux image, so the memory overhead would be large for efficient job migration.

Process hibernation saves an intermediate state of the OS booting procedure repeated in every OS booting to reduce the booting time [20]. However, hibernation saves all the processes at once, not suitable to save a single app state.

There is a research work for saving the Java VM state and migrating it [39]. It saves the state of the Java heap and the stack during the interpretation of a Java application. A similar technique would not be applicable to saving an app state though, because it does not deal with app-specific activities such as event handling.

Bellucci et al. [14] proposed to save the JavaScript and the DOM state using the JSON format for app migration. They solved the object reference alias problem similarly to ours. However, they provided no solution to save the closure variables, which occur frequently in real web apps and play an important role for data encapsulation, thus incomplete.

Lo et al. [15] completed the Bellucci's work by providing a solution to save the closure variables and event handlers. Their solution for closures is adding additional JavaScript code to create a

scope object and save the closure variable as a property of the object. Then, whenever a closure variable is updated in a closure function, the object property is also made to be updated with additional code. Finally, the scope object is saved as a property of the closure function, which allows retrieving the current value of the closure variable when the closure function is saved in the snapshot. Event handling is also solved using additional code which manages the events using some wrapper functions (otherwise there is no way to access the browser event queue). This instrumentation-based approach is different from ours which accesses the browser and the JavaScript engine internals. One problem of instrumented app, apart from the instrumentation and space overhead, is that the closure objects that the programmer wants to hide at runtime (they are supposed to exist only at the scope chain of the JavaScript engine) are exposed through the scope objects via global variables, causing a security issue.

There is also a research effort to restore a web app's state based on capture-and-replay [40]. They log all the events occurred during the app execution, so as to restore the app state by replaying the saved events one by one. This is useful for debugging and performance evaluation. However, restoration can complete only after all saved events are handled, so it would not be appropriate for app migration because the restoration overhead including the file size would be high, proportional to the running time before we capture. Logging and replaying is done by the client-side JavaScript, but is also done by a modified browser [41].

A limited form of snapshots has previously been exploited for reducing the start-up time of applications.

V8 JavaScript engine employs serialization, which saves the heap where the JavaScript built-in objects created during V8 initialization

exist [42]. Now the engine can start faster from the heap with the serialized built-in objects. This is for saving and restoring the initialized state of the JavaScript engine only. The Dart VM also allows saving the pre-parsed data for the classes or methods of Dart libraries and app script for faster app class loading [43].

Unlike V8 and Dart, our snapshot saves the app execution state including those objects created by executing the JavaScript code during app loading such as the framework objects or app objects. Also, we save a state beyond the VM, such as DOM objects, events, or event handlers.

For a Java VM (JVM) environment, romization has been used to serialize the loaded state of system classes including the class block, method block, and the bytecode [44]. This is compiled together with the JVM source code to reduce the class loading time when we launch an application.

There is also a research work to make a JVM process up in advance and to fork a JVM process from it to launch a Java application faster [45]. Actually, Android makes the Zygote process up in advance, which is a Dalvik VM process, where some of the system library classes and framework classes are loaded and initialized [46]. A new Android app is forked from the Zygote which accelerates app loading.

Most snapshot researches for Java or JavaScript VM restore the heap using the snapshot right when the VM starts. Once the heap is restored, it is impossible to add another snapshot of a different app to the existing heap. In fact, these VMs have one-app-per-one-VM execution model, so their snapshot is designed this way. On the other hand, web platforms allow multiple apps to run on a single JavaScript engine. However, V8 serialization is not designed to be addable to an

existing heap, either. That is, V8 also restore the heap using the snapshot right when the browser and the JavaScript engine starts. Once the heap is restored, it is also impossible to add another snapshot of a different app to the existing heap (even if an app-specific snapshot for the app were available). In our snapshot design, however, we can add the snapshot of an app to the existing heap when other apps are running on the same JavaScript engine. Our snapshot can correctly save those objects belonging to an app starting from its window object, and can restore them from the snapshot when other apps are running on the same browser. This is why we do not dump the whole JavaScript heap to the snapshot, but save each object one by one.

Chapter 7. Conclusion

Web platform is expected to become a viable app platform in a near future due to its advantage of portability and app productivity. As other app platforms, web platform is also expected to be available in diverse smart devices. One issue is how to benefit from these connected devices, installed with the same platform. The current wisdom appears to exploit the cloud service for sharing "static" digital contents such as pictures, audios, videos, and app binaries among those devices. One question is if the sharing can be extended to more of "dynamic" contents, more dynamic than PS4 Remote Play [47], AirPlay [22], or DLNA [48]which share at best the dynamic screen display of the static contents.

One good candidate that we propose to share is the app execution state. That is, if we share the app execution state among diverse devices so that the same app can be executed continuously and seamlessly, it would lead to a new user experience. This app migration will be simpler in the web platform due to its source code-based distribution and JSON-based data sharing between applications. We proposed a framework to save and restore the execution state of a web app with a snapshot based on the JavaScript code and the JSON strings. Unlike previous work with instrumentation, we save the state of an original app and access the browser internals to retrieve closures and events for correct saving. We showed app migration works for three apps.

The reason why we think app migration is valid is as follows. Since an app by definition is small unlike a full application, the program state to save would be small. Also, the data used by an app is often available in the server if they are huge (e.g., for a map app,

the map database itself is in the server, and the video/audio/picture files will be in the cloud server), and the app is just for the client-side display, so the client-side data state to save would also be small. So, app migration would requires a small overhead, yet it can give a new, interesting user experience. Apple Inc. has recently announced a similar idea called *handoff* for sharing apps between iPhone and iMac with new APIs [49].

There are still challenges left. One is accelerating the app migration, especially the app restoration. If the same web platform (CPU, OS, browser, JavaScript engine) is installed both at the source device and at the target device, faster restoration would be possible if we migrate the JavaScript objects in a binary form as they are in the heap, rather than the JSON-based JavaScript code, as we did for the launch-time optimization in Section 7. Space overhead would be higher, but not much, because some duplication in the JSON-based objects can be removed in the binary form (*e.g., var c ; function a() { function b() {..} c=b ;} a();* where JSON-based snapshot include duplicated function strings for *b* and *c*, while binary-based snapshot include a single function object pointed by both *b* and *c*).

There will be more issues to be solved to save the state correctly, especially for HTML5 objects such as Application Cache, Local Storage, Canvas, or Video. We think that the state of most apps even with these objects is still local to the client browser, hence being savable by our technique, using the existing APIs or additional new APIs to access the browser. These are left as a future work.

Another issues of web app is its performance. The time spent for app loading is important for the real-time behaviour of an app, but it cannot be easily reduced with an existing technique such as JITC. We proposed snapshot-based acceleration for app loading, which starts

the app from the snapshot saved in advance with the objects created. Due to the complex object representation in the JavaScript engine and the interaction with the DOM tree and the events, the snapshot requires an elaborate technique to save and restore, at the right time. With only saving JavaScript execution state, our evaluation shows the snapshot reduces the loading time tangibly for real apps by obviating the framework and app initialization.

We accelerated loading time of web app with additional optimization. One is saving the DOM state to the snapshot, which allows more JavaScript execution in the snapshot. We save DOM state by saving DOM interactions occur during web app loading as logs and replaying them later. Also, we reduced size overhead caused by snapshot. We compressed snapshot and implemented pre-decompressing snapshot which can hide decompression overhead and framework snapshot sharing which can share JavaScript framework state between web apps.

There should be various further researches which can improve loading time of web app based on snapshot. We can parallelize the restoration of the JavaScript objects and the DOM tree because they are independent, except that some DOM objects point the DOM nodes. This can reduce the restoration overhead of the app.

Bibliography

[1] "jQuery," [Online]. Available: http://www.jquery.com.

[2] "Enyojs," [Online]. Available: http://enyojs.com.

[3] "Ext JS," [Online]. Available: http://www.sencha.com/products/extjs.

[4] "Tizen," [Online]. Available: https://www.tizen.org.

[5] "webOS," [Online]. Available: http://www.openwebosproject.org.

[6] "Firefox OS," [Online]. Available: http://www.mozilla.org/en-US/firefox/os.

[7] "Adobe Flash runtime," [Online]. Available: http://www.adobe.com/products/flashruntimes.html.

[8] "HTML5," [Online]. Available: http://www.w3.org/TR/html5/.

[9] W. Ahn, J. Choi, T. Shull, M. J. Garzarán and J. Torrellas, "Improving JavaScript performance by deconstructing the type system," in *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, 2014.

[10] S.-W. Lee and S.-M. Moon, "Selective just-in-time compilation for client-side mobile javascript engine," in *In Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems (CASES '11)*, 2011.

[11] J. Oh, J.-w. Kwon, H. Park and S.-M. Moon, "Migration of Web Applications with Seamless Execution," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*, 2015.

[12] J. Oh and S.-M. Moon, "Snapshot-based Loading-Time Acceleration for Web Applications," in *In Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2015)*, 2015.

[13] "Offline Tetris," [Online]. Available: http://tetris.alexkessinger.net/.

[14] F. Bellucci, G. Ghiani, F. Paternò and C. Santoro, "Engineering JavaScript state persistence of web applications migrating across multiple devices," in *n Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '11)*, 2011.

[15] J. Lo, T. Kin, W. Eric and M. Ali, "Imagen: runtime migration of browser sessions for javascript web applications," in *In Proceedings of the 22nd international conference on World Wide Web (WWW '13)*, Republic and Canton of Geneva, Switzerland, 2013.

[16] "Sync tabs across devices," [Online]. Available: https://support.google.com/chrome/answer/2591582?hl=en.

[17] "Dropbox," [Online]. Available: https://www.dropbox.com.

[18] "JSON," [Online]. Available: http://www.json.org.

[19] Y.-Y. Su and J. Flinn, "Slingshot: deploying stateful services in wireless hotspots," in *In Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05)*, 2005.

[20] K. Baik, S. Kim, S.Woo and J. Choi, "Boosting up Embedded Linux device: experience on Linux-based," in *In proceedings of the Linux Symposium*, 2010.

[21] "Browser extension," [Online]. Available: http://en.wikipedia.org/wiki/Browser_extension.

[22] "Airplay," [Online]. Available: https://www.apple.com/airplay/.

[23] D. Lee, "JXON: an Architecture for Schema and Annotation Driven JSON/XML Bidirectional Transformations," in *In Proceedings of Balisage: The Markup Conference*, 2011.

[24] G. Wang, "Improving Data Transmission in Web Applications via the Translation between XML and JSON," in *In Proceedings of the 2011 Third International Conference on Communications and Mobile Computing (CMC '11)*, 2011.

[25] N. Nurseitov, M. Paulson, R. Reynolds and C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study," in *In Proceedings of CAINE*, 2009.

[26] "Serialization," [Online]. Available: http://en.wikipedia.org/wiki/Serialization.

[27] "JsonML," [Online]. Available: http://www.jsonml.org/.

[28] "Json.NET - Preserving Object References," [Online]. Available: http://james.newtonking.com/json/help/index.html.

[29] "JSON-Circular," [Online]. Available: https://github.com/StewartAtkins/JSON-Circular.

[30] "JSON-R: A JSON Extension That Deals With Object References (Circular And Others)," [Online]. Available: http://java.dzone.com/articles/json-r-json-extension-deals.

[31] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *In Proceedings of the sixth conference on Computer systems (EuroSys '11)*, 2011.

[32] P. Smutny, "Mobile development tools and cross-platform solutions," in *13th International Carpathian Control Conference*, 2012.

[33] "JavaScriptCore," [Online]. Available: http://trac.webkit.org/wiki/JavaScriptCore.

[34] "V8," [Online]. Available: https://code.google.com/p/v8/.

[35] "Pandaboard ES," [Online]. Available: http://pandaboard.org/content/pandaboard-es.

[36] "LZO," [Online]. Available: http://www.oberhumer.com/opensource/lzo/.

[37] "ODROID-XU4," [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825.

[38] M. Adiba and B. Lindsay, "Database snapshots," in *Proceedings of the sixth international conference on Very Large Data Bases*, 1980.

[39] T. Suezawa, "Persistent execution state of a Java virtual machine," in *In Proceedings of the ACM 2000 conference on Java Grande (JAVA '00)*, 2000.

[40] J. Mickens, J. Elson and J. Howell, "Mugshot: deterministic capture and replay for Javascript applications," in *In Proceedings of Networked Systems Design and Implementation*, 2010.

[41] B. Burg, R. Bailey, A. J. Ko and M. D. Ernst, "Interactive record/replay for web application debugging," in *In Proceedings of the 26th annual ACM symposium on User interface software and technology (UIST '13)*, 2013.

[42] "V8 snapshot," [Online]. Available: https://developers.google.com/v8/embed.

[43] "Dart Snapshot," [Online]. Available: https://www.dartlang.org/articles/snapshots/.

[44] A. Courbot, G. Grimaud and J.-J. Vandewalle, "Romization: Early Deployment and Customization of Java Systems for Constrained Devices," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2006.

[45] K. Kawachiya, K. Ogata, D. Silva, T. Onodera, H. Komatsu and T. Nakatani, "Cloneable JVM: a new approach to start isolated java applications faster," in *In Proceedings of the 3rd international conference on Virtual execution environments (VEE '07)*, 2007.

[46] D. Ehringer, "The Dalvik virtual machine architecture," March 2010.

[47] "PS4 Remote Play," [Online]. Available: https://support.us.playstation.com/app/answers/detail/a_id/5065/~/ps4-remote-play-and-second-screen.

[48] "DLNA," [Online]. Available: http://www.dlna.org/.

[49] "Apple Handoff," Apple Worldwide Developers Conference (WWDC), Jun 2014. [Online]. Available: https://developer.apple.com/videos/wwdc/2014/.

초 록

웹 어플리케이션(앱) 은 HTML5, CSS, 자바스크립트를 통해 구성된 앱으로서 소스코드 형태로 배포된다. 웹 앱은 브라우저가 설치된 디바이스에서 항상 실행가능하기 때문에 one-source, multi-platform 을 실현할 수 있도록 한다. 우리는 이러한 특성을 활용하여 새로운 형태의 UX 로서 앱 마이그레이션을 활용할 수 있을 것이다. 앱 마이그레이션은 스마트 기기간에 수행 중인 앱을 전송하는 것을 말한다. 우리는 이러한 마이그레이션을 웹 앱에 적용하여 수행중인 앱의 상태를 저장한 후에 다른 디바이스에서 그대로 이어서 수행할 수 있도록 하였다. 우리는 웹 앱의 수행을 스냅샷의 형태로 저장하였다. 스냅샷은 수행 상태를 저장하고 있어 이를 복원할 수 있는 다른 형태의 앱으로 볼 수 있다. 스냅샷에는 자바스크립트 변수와 DOM tree 가 JSON 포맷으로 저장된다. 이벤트 핸들러와 자바스크립트 클로저와 연관된 이슈를 브라우저와 자바스크립트 엔진 내부의 접근을 통해 해결하여 저장/복원이 가능하도록 하였다. 이를 통해 앱의 소스코드를 변경하지 않고 오리지널 앱 그대로 마이그레이션이 가능한 특징을 가진다. 마이그레이션 프레임워크를 Chrome 브라우저와 V8 자바스크립트 엔진에 구현하여 여러 앱들에 대하여 낮은 오버헤드로 마이그레이션 할 수 있었다. 또한 스냅샷을 통해 사용할 수 있는 최적화와 새로운 UX 에 대하여 논의 하였다.

웹 앱의 또 다른 이슈는 네이티브 앱에 대비해 낮은 성능이다. 웹 앱은 자바스크립트로 인해 성능에 제약이 있다. 자바스크립트는 동적 타입과 함수 객체, 프로토타입과 같이 효율적으로 수행할 수 없는 다양한 특징을 갖고 있는데 적시컴파일러를 사용하더라도 이를 효율적으로 개선하기 힘들다.

우리는 스냅샷을 활용하여 웹 앱을 향상시키는 새로운 방법을 제안하였다. 특히 로딩 타임을 향상 시킬 수 있는 기술이다. 일반적으로 웹 앱의 수행은 앱을 초기화하기 위한 웹 앱 로딩 과정을 거친 후에

이벤트 기반으로 수행된다. 만약 웹 앱의 로딩 중에 같은 동작을 반복한다면 특히 자바스크립트의 수행이 항상 같다면, 자바스크립트를 수행하는 대신에 미리 수행을 하고 자바스크립트 수행 상태를 스냅샷으로 저장하여 이를 통해 앱을 수행한다면 웹 앱을 빠르게 로딩할 수 있을 것이다. 웹 앱의 로딩은 jQuery, Enyo, Ext JS 와 같은 웹 프레임워크를 포함한다. 이러한 웹 프레임워크는 많은 자바스크립트 오브젝트를 생성하며 로딩 중에 앱 특유의 오브젝트를 생성하기도 한다. 만약 이러한 오브젝트들을 스냅샷의 형태로 저장하여 웹 앱의 초기상태를 저장하고 스냅샷으로부터 오브젝트를 복원하여 웹 앱을 로딩한다면 웹 앱의 로딩을 빠르게 할 수 있을 것이다.

자바스크립트 수행 상태를 저장한 스냅샷은 DOM 의 상태를 저장하지 않기 때문에 웹 앱을 성능을 향상하는데 제약이 발생한다. DOM 로그-리플레이 방식을 통해 동적으로 변화하는 DOM 상태를 저장하여 웹 앱의 로딩 타임을 더욱더 향상 하였다. 웹 앱의 로딩이 모두 완료된 직후의 상태를 스냅샷으로 저장할 수 있어서 로딩 중에 수행되는 모든 자바스크립트 수행을 제거할 수 있었다. DOM 로그-리플레이는 웹 앱의 로딩 중에 발생하는 DOM 관련 API 를 모두 로그로 저장하고 복원 시에는 저장한 로그를 다시 리플레이하여 DOM 의 상태를 복원한다.

우리는 이를 2 개의 자바스크립트 웹 프레임워크를 기반으로 한 웹 앱들을 대상으로 실험하였다. 전체 로딩 타임에 대비해 약 60%의 성능향상이 나타났고, 이는 실제 체감이 되는 성능이다.

스냅샷은 웹 앱의 수행 상태를 파일로 저장한 것으로서 메모리 오버헤드가 발생한다. 이는 IoT 같은 메모리가 부족한 디바이스 에서는 적합하지 않다. 그러므로 스냅샷의 사이즈를 줄이는 것이 필요하다. 첫 번째 방법으로 스냅샷을 압축하되 압축해제 오버헤드가 발생하지 않도록 스냅샷의 복원 전에 미리 다른 쓰레드에서 압축을 해제하는 선행 압축해제를 적용하였다.

두 번째는 자바스크립트 웹 앱을 개발하는데 많이 사용되는 자바스크립트 프레임워크를 웹 앱 사이에 서로 공유하도록 하는 프레임워크 스냅샷 공유를 개발하였다. 이를 통해 웹 앱 로딩타임은 1.7% 정도 느려지지만 메모리 오버헤드를 12x 배에서 2.59x 로 줄일 수 있었다.

주요어 : 웹 앱, 자바스크립트, 스냅샷, 웹 프레임워크, 로딩 타임, 앱 마이그레이션
학 번 : 2009-20839