Snapshot-based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution

JiHwan Yeo
Seoul National University
Seoul, Korea
jhyeo@altair.snu.ac.kr

ChangHyun Shin
Seoul National University
Seoul, Korea
schyun@altair.snu.ac.kr

Soo-Mook Moon
Seoul National University
Seoul, Korea
smoon@snu.ac.kr

ABSTRACT

JavaScript execution is heavily used during the loading of web apps, taking a substantial portion of the app loading time. To accelerate JavaScript execution, *snapshot-based app loading* has been proposed [5, 17]. We take a *snapshot* of the JavaScript objects in the heap at some point during app loading (which we call *snapshot point*) and save them in a file in advance. Then, we start app loading by copying the objects in the snapshot to the heap directly, skipping JavaScript execution to create those objects. One issue is that the JavaScript execution state at the snapshot point should be the same at every app loading. If JavaScript execution included in the snapshot is involved with some nondeterminism (e.g., use random function or current time/location), snapshot-based app loading might be inapplicable since the loaded state might differ each time.

In this paper, we perform an empirical study for the nondeterministic behavior of web apps during app loading. The result shows that nondeterminism is used frequently, but its impact is small from the user's perspectives. Based on this observation, we propose two techniques that can increase the applicability of snapshot. First, we propose two types of snapshot point, which allows saving as much execution state as possible in the snapshot, just before the nondeterministic execution affects the user's perception. Second, we develop a tool to identify those snapshot points by static and dynamic analysis, so that the developer can easily identify them. Our experimental results show that the techniques can accelerate the app loading by 1.81 times, by applying the snapshot that would otherwise be inapplicable, achieving a performance comparable to that of a snapshot that completely ignores the nondeterminism issues.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**; *Runtime environments*; *Scripting languages*.

KEYWORDS

Web app; JavaScript engine; Snapshot; Nondeterminism

ACM Reference Format:
JiHwan Yeo, ChangHyun Shin, and Soo-Mook Moon. 2019. Snapshot-based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution. In *Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3308558.3313575

1 INTRODUCTION

Web apps are popular because they are extremely portable, running on any device with a web browser. They are standard apps for commercial web platforms such as Tizen [11] or WebOS [8], being employed by embedded devices such as smart TVs, smart watches, or IoT devices.

In the mobile environment, loading time is an important factor affecting the user experience and business revenues [3, 4, 25]. For example, it is found that every 100ms of latency costs 1% reduction of sales for Amazon [10]. Reducing the loading time is also important for web apps, especially in the embedded devices such as smart TVs, since their limited hardware resources might lead to a long loading time, sometimes unacceptable to the users.

One technique to accelerate app loading is *snapshot* [17]. Since app loading usually produces the same execution state, especially the same JavaScript state, we can save the loaded state in a file called the snapshot in advance and perform app loading by copying the state from the file, instead of executing the JavaScript code to produce the same state. Recently, Google's Chrome browser introduced a technique called the *custom snapshot* [5], extending its previous technique *serializer*, which does the same.

Snapshot has a subtle correctness issue. The loading process of a web app may include various *nondeterminisms*. For example, it can executes a JavaScript function *Math.random*, which returns a random value, or a function *Date*, which returns the current time. Also, it can execute Web APIs that retrieve the device status or the browser information. If the results of these nondeterministic executions were also saved in the snapshot, app loading using a snapshot created by a previous run might not produce a correct state for the current run. Existing snapshots include only those JavaScript executions that are deterministic [5, 17]. For this, the app developer must specify a proper snapshot point, up to which only deterministic JavaScript executions are made.

Not all nondeterministic JavaScript executions are unqualified for snapshot, though. Each nondeterminism has a different impact on app loading, and some nondeterminisms do not show any difference from the user's perspectives. For example, the *jQuery* web framework uses *Math.random* to generate a unique identifier and use it as a property name [6]. The property name will change at each loading, but it is used only inside the framework. So, changing the property name does not affect the DOM tree displayed on the screen or the behavior of the event handlers. It is completely safe to include these nondeterministic executions in the snapshot, thus increasing performance without affecting correctness.

Deciding the best snapshot point that excludes unsafe nondeterminism but includes as much JavaScript execution as possible would be a challenge. It is difficult for the app developer to analyze the app to find all variables affected by nondeterministic executions and all affected execution states. If an app uses a JavaScript framework, the developer should fully understand the framework code, which is not easy for most developers. In fact, even the snapshot of the commercial Chrome browser does not give any information on nondeterminism for JavaScript execution [5], so the developers have to analyze by themselves or go through a trial and error method to choose a proper snapshot point.

This paper made three contributions. First, we perform an empirical study of the nondeterministic behavior of web apps at loading time. The result shows that nondeterminism is frequent, but its impact is often little from the user's perspectives, as the unique identifier case of jQuery.

Secondly, we propose two nondeterminism-aware snapshot points that can lead to a higher performance than the snapshot with deterministic executions only. They allow the developers to choose between performance and correctness. The first snapshot point is *deterministic DOM*, which includes the JavaScript executions up to the point where the DOM tree is deterministic. This will include the JavaScript execution that does not affect the user's perception. The second snapshot point is *semi-randomized DOM*, which additionally saves the JavaScript executions that produce a nondeterministic DOM tree with *Math.random*. One example is a game app showing a random initial screen when loaded. If we make a snapshot including *Math.random*, the app will show the same screen at each loading. This does not affect the correctness, but only the user's perception (e.g., a minesweeper app shows the same position of mines at each loading). To handle this issue, we create multiple snapshots and use one of them randomly at loading time to mimic a randomized execution.

Finally, we create a tool to find the snapshot point for web apps using static and dynamic analysis to increase the applicability of the snapshot. We instrument HTML files and JavaScript code, and the browser runs the instrumented app to collect information such as JavaScript variable read/write and DOM tree changes. The tool creates a JavaScript dependency graph based on the information to find all JavaScript statements affected by nondeterminism. If such a JavaScript statement changes the DOM tree, it is regarded as a nondeterministic DOM change. Our tool identifies all script tags or event handlers that make nondeterministic DOM changes so as to help the app developer to decide a proper nondeterminism-aware snapshot point.

Previous studies on nondeterminism classified the source of nondeterminism in the web pages, and made the execution deterministic by saving the execution results of nondeterminism and using them in subsequent executions [2, 14]. Snapshot also saves the results of nondeterministic executions and reuse them for acceleration, but we identify nondeterminisms that affect little the user's perception and save only those executions in the snapshot.

Our experimental results show that the proposed techniques can accelerate the app loading by 1.81 times on average compared to a snapshot with deterministic executions only, by including safe, nondeterministic JavaScript executions in the snapshot that would otherwise be not includable. This achieves a performance comparable to that of a snapshot that completely ignores the nondeterminism issues.

The rest of the paper is as follows. Section 2 reviews snapshot-based app loading. Section 3 analyzes the nondeterminism sources in web apps. Section 4 proposes nondeterminism-aware snapshot points. Section 5 shows the nondeterminism analysis tool. Section 6 depicts the implementation and Section 7 shows the evaluation results. Related work is in Section 8 and conclusion is in Section 9.

2 SNAPSHOT-BASED ACCELERATION OF WEB APPS

The browser loads a web app by executing its HTML code, embedded with the script tags of JavaScript global code. It parses the HTML code to a DOM tree. When a script tag is met, the JavaScript code is executed after creating a *window* object in the heap. After HTML parsing completes, the browser fires *DOMContentLoaded* event and *onload* event, and the JavaScript event handlers registered in the script tags will be executed. This completes the app loading process, and then the app will work in an event-driven manner based on the user events.

Snapshot-based acceleration is based on an observation that app loading often repeats the same initializations. For example, most web apps employ JavaScript frameworks such as jQuery [6], which are initialized during app loading. Similarly, the app itself is initialized as well. These initializations are done by executing the same JavaScript code, which often produces the same objects in the JavaScript heap. So, instead of executing the JavaScript code to create those objects at each loading time, it is better to save the JavaScript objects right after these initializations to a file in advance, and to start an app by copying the JavaScript objects from the file to the heap directly [5, 17]. Even if we need to save or restore the objects one-by-one, starting from the JavaScript *window* object, snapshot-based acceleration is shown to be around four times faster than executing the JavaScript code [17].

Actually, the loading state of a web app includes the DOM state as well as the JavaScript state. Unlike the JavaScript state which can be saved as above, saving the DOM state is not straightforward. For the initial snapshot works [5, 17], the DOM tree is not saved in the snapshot, but completely re-built using the usual HTML parsing. One issue is that if some JavaScript code executed during app loading changes the DOM tree (e.g., add/delete some DOM nodes), such changes will be lost when employing snapshot-based app loading since the snapshot includes only the JavaScript objects. This constrains the snapshot point, such that only the JavaScript executions that do not change the DOM tree are eligible for inclusion in the snapshot [5, 17]. More precisely, since we can have only a single snapshot file, we can save a "consecutive" sequence of JavaScript executions until the first execution that changes the DOM tree is met, which limits the coverage.

More recent snapshots save some of the DOM state as well [16, 29]. One method records the log of all the DOM APIs that JavaScript executions invoke until the snapshot point [16]. Snapshot-based app loading re-builds the DOM tree using HTML parsing as previously, but restores the correct DOM tree state by replaying the DOM logs, while restoring the JavaScript state from the snapshot at the snapshot point. Another method is saving the whole DOM tree as a serialized HTML string using the *innerHTML* API at the snapshot point [29]. Snapshot-based app loading re-builds the DOM tree, but at the snapshot point, it is overwritten by the de-serialized tree from the HTML string. For the implementation of nondeterminism-aware snapshot points in this paper, we employed the first implementation.

One advantage of saving the DOM state in the snapshot is that we can take a snapshot at a later point of app loading, even after the entire app loading process completes [16], thus increasing the applicability of snapshot. Generally, JavaScript executions during app loading comes from the framework script tags, the app script tags, and the event handlers, in this order. For those snapshots that cannot save the DOM state [5, 17], a snapshot can be taken just before the first DOM-changing JavaScript (actually, the first DOM-changing script tag since they cannot take a snapshot at the event handler). If we save the DOM state, snapshot-based app loading can skip the executions of all script tags and event handlers up to the snapshot point, irrespective of whether they change the DOM tree or not, then restore the JavaScript state from the snapshot, thus performing better.

One issue is that those JavaScript executions included in the snapshot should be deterministic, meaning that they should produce exactly the same execution state every time. If they are nondeterministic, the state stored in the snapshot might be different from the actual app loading state. Previous snapshots did not consider nondeterminism [5, 17], or even if they do, they let the app developer to decide the snapshot point manually [16, 29]. According to our study about nondeterminism in Section 3, however, there occur lots of nondeterminisms during app loading, so we need a better way to apply snapshot considering nondeterminism.

3 NONDETERMINISM IN LOADING OF WEB APPS

In this section, we examine nondeterminisms during app loading and list up the sources of nondeterminism. We also analyze how app loading exploits nondeterminism and examine the impact on the DOM tree.

3.1 Sources of Nondeterminism in App Loading

There are previous researches that analyze the source of nondeterminism that can occur in web apps [2, 14]. Unlike them, we are only interested in nondeterminism that may occur during app loading. For example, many events during app execution are involved with nondeterminism, but only the timer event is source of nondeterminism in the app loading process (see Section 3.1.3).

We have classified the sources of nondeterminism as *nondeterministic functions*, *web APIs*, and *interrupts*. Among the JavaScript functions, those declared in the HTML5 specification are categorized as *web APIs*, and the remaining functions are classified as *nondeterministic functions. Interrupts* mean the nondeterminism caused by events such as timer events. Details are discussed below.

3.1.1 Nondeterministic Functions. Among the functions in the JavaScript specification, there are two nondeterministic functions: *Math.random*, which returns a random value, and *Date*, which can be used to return the current time, thus being nondeterministic.

3.1.2 Web APIs. JavaScript supports a variety of APIs for browsers. The most commonly used APIs are related to the DOM tree, allowing the developers to change the display by adding or removing elements from the DOM tree. DOM is standardized and supported by all browsers, but latest APIs are supported by a few browsers. In this paper, we analyze only widely-used APIs, supported by the mainstream browsers. We have classified web APIs into three types.

File system is a set of APIs used to read and write persistent data in the browser, such as cookie, localStorage, or IndexedDB. We use the APIs to read preferences or data, thus being nondeterministic depending on their existence.

Device information is the APIs that fetch the status of the current device. They allow the developers to get device information from the web apps. Among many APIs, we consider geolocation, device orientation, and device motion APIs that are supported by most web browsers. The geolocation API uses GPS or IP address to determine the user's current location, and it is used by apps such as maps and weather to show information based on the location.

Browser Information is an API that retrieves the information of the browser running the app. Developers can get the size/position of the browser using those APIs. The userAgent API is widely used to get the type of the browser and its version. For snapshot implementation, the same browser is used, so we exclude userAgent from our analysis.

3.1.3 Interrupts. Previous researches have analyzed the events in detail [2, 14], but the events that can occur during the app loading process are limited. Generally, events can be triggered by a user interaction, by a JavaScript function call, or by the web browser. In the middle of app loading, the DOM element is not displayed on the screen because the rendering is not completed yet. Since it is difficult for the user to activate an event during loading, the event triggered by the user cannot be the source of nondeterminism.

JavaScript execution can fire an event directly by using the *dispatchEvent* function or by registering a timer event with the *setTimeout*, *setInterval* function. If *dispatchEvent* is used in the JavaScript code, the execution order is fixed because the event occurs immediately after the execution of the code, thus being deterministic. On the other hand, timer events can be nondeterministic. *setTimeout* and *setInterval* are used to register timer events which will be fired after a certain amount of time. Consider two consecutive script tags where the first tag registers a timer event that occurs 2.0 seconds later while the second tag registers a timer event that occurs 1.0 seconds later. If the first tag executes for 0.5 seconds, the first event will fire at 2.0 second while the second one will fire at 1.5 (0.5+1.0) second. If the first tag executes for 1.5 seconds due to some delay (e.g., page miss), however, the first event will still fire at 2.0 second, but the second one will fire at 2.5 (1.5+1.0) second. So, the execution order of the events can be nondeterministic.

Only two events are fired by the browser during app loading: *DOMContentLoaded* event and *onload* event. The *DOMContentLoaded* fires when the HTML parsing is done, and the *onload* event fires when the resources in HTML are completely downloaded. In most apps, the event handlers executed for each of these two events and their execution order are fixed, thus being deterministic.

Table 1: Number of nondeterminism occurrences during app loading

Type	Nondeterministic Functions		File System APIs		Browser Information	Interrupts		Total
	Random	Date	Cookie	localStorage	Size	setTimeout	setInterval	
Framework global code	0/6	0/12	0/0	0/0	0/0	0/0	0/0	0/18
App global code	251/251	1/1	0/0	0/0	9/9	0/1	0/0	261/262
Event handler	582/583	57/64	3/3	3/3	0/0	0/2	0/3	645/658

3.2 Nondeterminism Usage at Web App Loading

We used a dynamic analysis tool *Jalangi* [21] to record all nondeterminisms that occur during app loading. We observe every *function call* and *property access* during app loading, where nondeterministic result might occur (e.g., *Math.random()* or localStorage[‘setup’]). We identify the location and the number of occurrences for nondeterminism defined in Section 3.1. To examine the impact of each nondeterminism on the user’s display, we manually analyze the effect on the DOM tree, which is what the user see.

For this study, we selected 9 web apps that make extensive use of JavaScript. We selected these apps from the JavaScript projects on GitHub, and sample apps from the official framework site. Six of them use *jQuery* framework [6] and the rest use *enyo* framework [7]. jQuery is widely used by 73.3% of all websites and enyo is used on the commercial WebOS platform [26].

Each source of nondeterminism has a different impact on app loading. Some nondeterministic value is assigned to a local variable, thus not affecting the global state such as the DOM tree, while others can be used to affect the DOM tree. A previous study has classified nondeterminism based on whether it affects the global state [22]. For example, a nondeterministic JavaScript execution creating a unique identifier affects the global state, but it does not affect the user’s display. We also categorized nondeterminism based on if it affects the DOM tree, thus changing the user’s display and making the user feel the difference. For *nondeterministic functions* and *web APIs*, we analyzed if the DOM tree changes when the return value changes. For *interrupts*, we checked whether the DOM tree changes when the execution order of event handlers is changed. We also categorized nondeterminism based on where it occurs: *framework global code* in the framework script tag, *app global code* in the app script tag, or *event handlers* of the *DOMContentLoaded* and *onload* events.

Table 1 shows the total number of nondeterminism occurrences during the loading of 9 apps. The left value of each cell is the number of nondeterminisms that affects the DOM tree, while the right value is the total number of occurrences. Some nondeterminisms such as *IndexedDB* and *browser position* are not used, so they are not shown in the table. In the app loading process, nondeterminism occur 938 times in total, and the average number of usages per app is 104. If we categorized nondeterminism based on where it occurs, we can see that the framework global code, app global code, and event handlers use nondeterminism 18 times, 262 times, and 658 times, respectively. Framework global code creates objects and registers event handlers to initialize the framework, and those

(a) Unique identifier generation

```
jQuery.expando =
  "jQuery" + (version+Math.random()).replace(/\D/g, "");
Sizzle.expando = "sizzle" +-(new Date());
```

(b) Timestamp creation

```
jQuery.now = function() {
  return +( new Date() );
};
this.timeStamp = src && src.timeStamp || jQuery.now();
```

Figure 1: jQuery example code with nondeterminism sources

tasks are mostly deterministic. App global code and event handlers heavily use nondeterminism to change the DOM tree by reading the nondeterministic data such as the setting of the app or the current time. Since many jQuery apps uses event handlers to change the DOM tree [13], there are many occurrences of nondeterminism in the event handlers.

Among the sources of nondeterminism, the most frequent calls were *nondeterministic function*, among which *Math.random* is dominant. Since apps often use *Math.random* to determine the location and type of many DOM elements, so it is called frequently.

We examined the effect of nondeterminism on the DOM tree and found that none of the 34 nondeterminisms in the framework global codes affect the DOM tree. The reason is that the framework code uses nondeterminism in a way of not affecting the DOM tree. For example, the jQuery framework global code calls *Math.random* once and *Date* twice, as illustrated in Figure 1. Figure 1 (a) uses *Math.random* and *Date* to create a unique identifier, which is often used for avoiding conflicts between variable names when using multiple versions of jQuery in an app. Figure 1 (b) uses *Date* to record a timestamp, often for an event object to tell when the event is generated. This is useful for profiling the event performance, by measuring the time taken between two events generated as a result of the user interaction.

Since the unit of JavaScript execution included in the snapshot is a script tag or an event handler, we categorized nondeterminisms as a unit of script tags and event handlers. Table 2 shows the number of script tags and event handlers that have nondeterminism during the loading of 9 apps. In the framework script tag, 6 out of 11 are nondeterministic, but none of them affect the DOM tree. In the app script tag, only 4 out of 12 have nondeterminism, but all of them affect the DOM tree. Most of the event handlers (7 out of 9) have

Table 2: Number of script tags and event handlers

Type	Nondeterministic & affecting DOM	Nondeterministic	Total
Framework tag	0	6	11
App tag	4	4	12
Event handlers	6	7	9

nondeterminism, and most (6 out of 7) affect the DOM tree. We also found that nondeterminism is used extensively for some script tags and event handlers, but it is not used at all in others. For example, only two game apps (breakdom and piratepig) call *Math.random* 781 times. Those apps have more than a hundred DOM elements and use *Math.random* to specify the type of DOM elements or position of DOM elements randomly.

In summary, nondeterminism occurs frequently during app loading, but there are many scrip tags and some event handlers that are deterministic, or are nondeterministic yet not affecting the DOM tree. Based on this observation, we propose two nondeterminism-aware snapshot points that minimize the impact on the user's display while achieving a high performance.

4 NONDETERMINISM-AWARE SNAPSHOT POINTS

JavaScript execution order during app loading is fixed, composed of the executions of the framework script tags, app script tags, event handlers for the *DOMContentLoaded* event, and the event handlers for the *onload* event. We will decide a snapshot point at the end of some script in this sequence of scripts, where we create and restore snapshot.

If we consider nondeterminism when we choose a snapshot point, there can be two extremes. One is ignoring the effect of nondeterminism. Since we can save the DOM state in the snapshot [16, 29], this allows us to save all JavaScript executions in the snapshot, thus taking a snapshot point at the end of app loading. We define this snapshot point *nondeterministic JS*, which saves all JavaScript executions without considering nondeterminism.

The other extreme of snapshot is including deterministic JavaScript executions only, which we define *deterministic JS*. This method is consistent with the basic idea of a snapshot that stores and reuses the exactly same result of repeated JavaScript execution. However, as shown in Table 2, 17 out of 32 script tags and event handlers use nondeterminism, and more notably, 54% of the framework script tags have nondeterminism. Since the framework script tags tend to precede other script tags (i.e., the app global code uses the functions defined in the framework global code), if we cannot include some framework script in the snapshot due to nondeterminism, we cannot include any of other subsequent script tags and event handlers, either. This would take a snapshot point early in the app loading process, reducing the coverage of the snapshot. As such, the performance improvement using a snapshot of *deterministic JS* would be seriously limited.

To compromise between performance and correctness that the user perceives, we propose two more snapshot points considering nondeterminism. The first one is *deterministic DOM*, which saves all (even nondeterministic) JavaScript executions as long as they make the DOM tree deterministic. This includes creating unique identifiers using *Math.random* and logging the timestamps in Figure 1. From the user's perspectives, an execution is deemed correct if the result it produces is acceptable to the user [9]. In the web environment, the user observes only the DOM tree and the behavior of the event handlers, so the user will think that snapshot-based app loading is correct if those two states are deterministic. Usually, nondeterminism that affects the event handlers also changes the DOM tree, so we consider only the DOM tree in this paper (we discuss the event handler issue later in Section 5.3). Using a snapshot point of *deterministic DOM*, we can get additional performance by including more JavaScript executions than *deterministic JS* without affecting the user's display.

The second snapshot point is *semi-randomized DOM*, which extends *deterministic DOM* by saving additional JavaScript executions that may change the DOM tree using *Math.random*. Unlike *deterministic DOM*, the DOM tree of snapshot-based app loading can be different from that of regular app loading, but the correctness is not actually compromised. That is, if an app uses *Math.random*, any return value between 0 and 1 is correct, and the snapshot would include one of the possible, correct values (this is not true if a function retrieves the time or the location, since there should be only one correct value). For example, a game app is supposed to show a random, initial screen on each app loading (e.g., a minesweeper game with different mine locations generated by *Math.random*). If such a JavaScript execution is included in the snapshot, the initial screen would be fixed (e.g., the same location of mines), which might look odd, but still correct. *Semi-randomized DOM* exploits this randomness and save the executions that randomly change the DOM tree in the snapshot. Users might feel that the execution is incorrect if the state is fixed in one state, so we created multiple snapshot files generated by multiple app loadings, and use one of them randomly. If we store enough snapshots and use them, users might not feel any difference. This snapshot point is effective for our game apps that change the DOM tree each time of app loading using lots of randomness. For example, breakdom and piratepig app uses *Math.random* 212 and 560 times, respectively. *Semi-randomized DOM* saves those JavaScript executions in the snapshot and archives a substantial performance improvement. This approach leads to space overhead, but users can get a significant performance gain, as will be seen in the experiment.

5 NONDETERMINISM ANALYSIS TOOL

Previous section proposed two nondeterminism-aware snapshot points in the app loading process. Unfortunately, it is difficult for the developers to identify them since they must know every source of the nondeterminism in the app code as well as JavaScript variables and DOM tree elements affected by nondeterminism. In fact, most apps use one or more JavaScript frameworks and it is almost impossible for the developers to understand them completely since the frameworks are substantial with many features. To handle this problem, we propose an analysis tool, which informs the snapshot point automatically by analyzing nondeterminisms that occur during app loading using static and dynamic analysis.

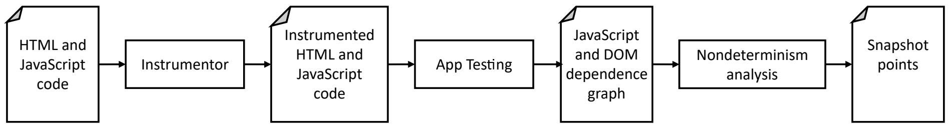


Figure 2: Overview of our nondeterminism analysis tool

5.1 Overview of the Tool

Figure 2 overviews our nondeterminism analysis tool, which uses three steps to identify the snapshot points. First, the tool statically instruments the HTML codes and the JavaScript codes to extract the logs of read/writes for the JavaScript variables and of the DOM mutations in app loading. It also records the sources of nondeterminism. Secondly, we perform app loading with the instrumented app, dynamically collecting the logs, and our instrumented event handler invoked at the end of app loading uses those logs to generate a dependency graph of JavaScript and DOM. We actually run the instrumented app multiple times and get multiple dependency graphs. Since nondeterminism can be used at the conditional statement, the control flow of the app can change, producing a different dependency graph. So we run the app multiple times to deal with diverse control flows. Third, the tool uses the dependency graph to analyze if a DOM change during app loading is affected by a source of nondeterminism. Based on the analysis, it finds which script tags and event handlers can be included for the two proposed snapshot points. Then, the app developer will decide a proper snapshot point considering both their correctness policy and the expected performance benefit.

5.2 Nondeterminism Analysis

Our instrumentor is implemented by extending JS-Slicer [28], which instruments the HTML/JavaScript code to produce a runtime dependency graph for JavaScript and DOM. In the dependency graph, each node represents a statement, and an edge *(i, j)* represents a dependency relationship between node *i* and *j*. JS-Slicer uses static analysis to extract the control dependency and dynamic analysis to identify the JavaScript and DOM dependency.

We modified JS-Slicer to record the usages of nondeterminism based on the source of nondeterminism listed in Section 3.1. We check every *function call* and *property read* at app loading time to see if any source of nondeterminism is used. If nondeterminism is used, the *id* of the statement and the *type* of nondeterminism in the statement (e.g., random) is recorded in the *nondeterminism map*. We use the map information to find the snapshot points later.

After running the app, we analyze if a script (script tag or event handler) in the script sequence of app loading includes nondeterminism, and if so, analyze whether the nondeterminism affects the DOM tree or not. So, for each DOM mutation statement, we compute all statements whose execution result is "*reachable*" to it. If a nondeterministic statement is reachable to a DOM mutation statement, the script that includes the nondeterministic statement "*affects*" the DOM tree. It should be noted that a nondeterministic statement can affect a DOM mutation statement in a later, different script in the script sequence.

Figure 3 shows an example of nondeterminism analysis for a script tag. Figure 3 (a) is a JavaScript code that declares two variables and displays the values in the DOM tree. Figure 3 (b) shows the corresponding dependency graph and the nondeterminism map. We create a node of the dependency graph for each statement, and the number next to each node represents its *id*. Since the return value of node 1 is assigned to the variable *n1* in node 2, node 2 has a dependence on node 1. To analyze whether the DOM changes in the script tag are affected by nondeterminism, we find all the reachable nodes to node 4 which has a DOM tree mutation. In this case, node 1, 2, and 3 are reachable to node 4, so we check the nondeterminism map to see if there are nondeterminism among those three nodes. Since node 1 is in the nondeterminism map, and its type is random, we record that this script tag has nondeterminism that affects the DOM tree, and its nondeterminism type is random.

There can be an if-statement in a script whose execution flow depends on the result of a nondeterministic statement such as *nondeterministic functions*, *device info*, or *interrupts*. In this case, we make an edge from the nodes located in the true and false paths of the if-statement to the nondeterministic statement in the dependency graph, based on the control dependence result obtained during static analysis. This is for computing nondeterministic scripts that may affect the DOM tree conservatively.

Even with a conservative dependency graph, there might be a case that might miss a nondeterministic script that affects the DOM tree. For example, consider a DOM tree element defined before an if-statement that is affected by *Math.random*. If it is redefined only in the true path of the if-statement, *Math.random* would appear not affecting the DOM tree if the false path is taken during app loading with the instrumented app, although it is evident that the DOM tree is changed depending on *Math.random*. To handle this case, we run app loading multiple times to cover diverse control flows, hence diverse dependency graphs.

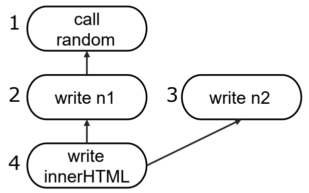
There is a more subtle case. Consider an app whose loading uses a file system API that checks if a file exist, and if so, updates the DOM tree with the file (so the API is nondeterministic). The problem is that the file can be created only by an event handler executed during an user interaction after app loading completes. For a *note* app, for example, if the user writes a new note after current app loading, and the app should show the saved note at the next app loading. Even when we run the instrumented app loading multiple times, if we do not fire the event and execute the event handler after app loading, the API will always return false, thus classified not affecting the DOM tree. So, if an app uses a file system API, we need to collect the information on registered event handlers and fire the user events a few times even after app loading completes.

Based on the above analysis, we identify the scripts to be included in each type of snapshot point. Algorithm 1 shows the algorithm to find a script where we create a snapshot for a given type of snapshot point. *checkCandidate* function analyzes each script

(a) JavaScript code

```
var n1 = Math.random();
var n2 = 10;
document.body.innerHTML = n1 + n2;
```

(b) Dependency graph & Nondeterminism map



Nondeterminism Map

ID	Type
1	random
...	...

Figure 3: Nondeterminism analysis for an example JavaScript code

Algorithm 1 Snapshot Point Detection

```
function checkCandidate(script)
    if script has nondeterminism then
        if nondeterminism affect DOM tree then
            if only random affect DOM tree then
                return "semi-randomized DOM candidate"
            else
                return "nondeterministic JS candidate"
        else
            return "deterministic DOM candidate"
    else
        return "deterministic JS candidate"

function findSnapshotPoint(snapshot_point_type)
    $snapshot\_point \leftarrow \emptyset$
    for i = 1 to number of scripts do
        $candidate \leftarrow checkCandidate(script_i)$
        if $snapshot\_point\_type \subset candidate$ then
            $snapshot\_point \leftarrow script_i$
        else
            return $snapshot\_point$
    return $snapshot\_point$
```

and tells which candidate snapshot point it can possibly belong to. Since a *deterministic JS* is the strictest candidate, it can also be included in a more relaxed candidate such as *deterministic DOM*, *semi-randomized DOM*, and *nondeterministic JS* snapshot points. Similarly, a *deterministic DOM* candidate can also be included in *semi-randomized DOM* and *nondeterministic JS* snapshot points. Finally, a *semi-randomized DOM* candidate can be included in *nondeterministic JS* as well.

findSnapshotPoint function checks all the scripts to be executed during app loading and returns a script where we need to take a snapshot point for a given type. It checks sequentially for each script in the app loading process if it can be a candidate for a given type of snapshot point based on the result of *checkCandidate*. If so, the script can be a snapshot point but we continue to check if we can extend the snapshot point to the next script (by iterating the loop in Algorithm 1). Otherwise, the script is the first one that cannot be included in the snapshot, so we return the current snapshot point, which is the previous script.

5.3 Limitation

To decide a proper nondeterminism-aware snapshot point, we analyzed if nondeterministic scripts executed during app loading affect the DOM tree. However, they can also affect the behavior of event handlers executed after app loading for some apps. For the nondeterminism analysis tool to cover the behavior of these event handlers, the tool should record a dependency graph for them as well. Unlike the scripts executed during app loading, the execution order of these event handlers would be arbitrary, depending on user interaction. So, we should trigger every sequence of user events and check the event handler is affected by nondeterminism in app loading. It is challenging to test event sequence using dynamic analysis only, so using some static analysis approach for the event handler would be effective [12], which is left as a future work.

For the apps using *file system* APIs, we need to trigger the events for user interaction to explore more paths of app loading scripts, as mentioned in Section 5.2. We currently trigger user events up to a fixed depth of two. App testing time would dramatically increase if we use a higher depth for more precise analysis. In the field of automatic testing, there are researches to minimize the triggered events [1, 23], which we can use for efficient testing in the future.

6 IMPLEMENTATION

We extracted dependencies between JavaScript variables on the top of JS-Slicer [28], which performs dynamic slice using JavaScript. For nondeterminism analysis, we update the shadow execution rule of the existing implementation to extract information of nondeterminism. JS-Slicer calculates the dependency for the DOM tree, but only using the *appendChild* and *innerHTML* among the APIs that can modify the DOM tree. The *MutationObserver* API is a Web API that monitors every DOM tree changes [24]. We used the API to track the DOM tree changes after the execution of the *property write* and *function call* statements.

JS-Slicer correctly generated dependency graphs for most JavaScript implementations, but do not for some native functions. JS-Slicer assumes that all arguments have a data dependency on the return value. However, in a native function such as *Array.sort*, we must attach additional edges to the dependency graph. *Array.sort* function compares the elements in an array and sorts them in order. This function compares two values using the function at the first argument. If the function that compares the elements uses nondeterminism, the array is randomly sorted and there must be a dependency between the return value of the function argument and the base array. We analyzed 28 methods of *Array* object. 10 of them update the array after the function call, but JS-Slicer creates no dependency between the argument and the base array. 7 of them need dependency between the return value of the argument and the base array as *Array.sort*. *Array.sort* is included in both case and we add dependencies on those 16 methods.

For automated testing, we use selenium to run and trigger events [20]. We run the app without file system five times and trigger user events up to a fixed depth of two for others. Finally, we create five snapshot files for *semi-randomized DOM* to simulate randomized execution.

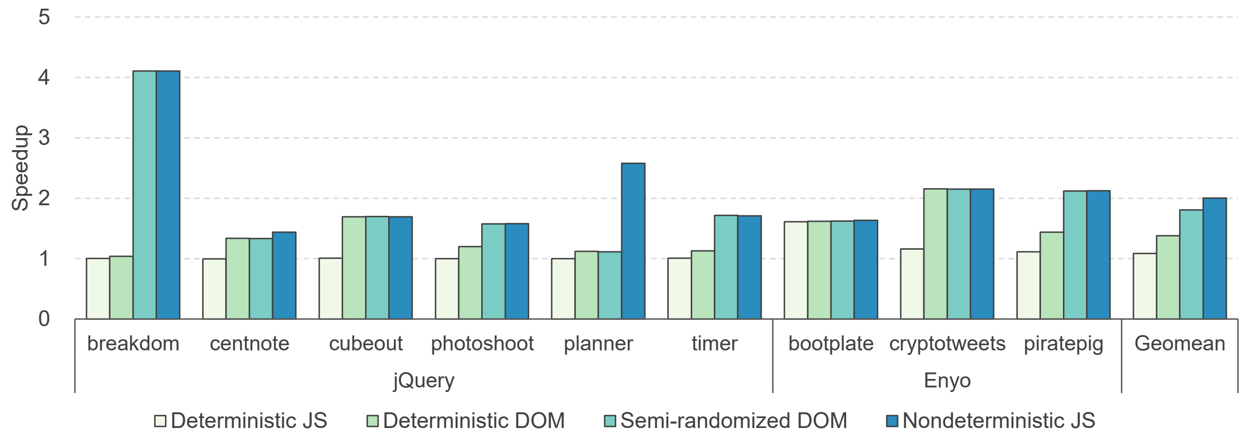


Figure 4: Snapshot-based app loading speedups of four snapshot points compared to original app loading

7 EVALUATION

We measured the performance of app loading based on the four snapshot points obtained from the nondeterminism analysis tool, and examined the effect of nondeterminism in the web app. Our snapshot is implemented based on the WebKit browser (rev-149728) and executed on a Pandaboard ES with 1.2GHz ARM Cortex-A9 CPU and 1GB of RAM. We experimented with 6 jQuery apps and 3 enyo apps used in the analysis in Section 3. We used a nondeterministic analysis tool to find *deterministic JS*, *deterministic DOM*, and *semi-randomized DOM* snapshot points. We also compare with the *nondeterministic JS* snapshot point, which saves the result of the whole JavaScript executions during app loading in the snapshot.

7.1 Snapshot Point Performance

Figure 4 shows the speedup of app loading time for the four snapshot points, compared to the app loading time without the snapshot. *Deterministic JS* accelerate app loading by 1.09 times on average. Web apps in our experiment use the JavaScript framework (jQuery or Enyo). jQuery framework calls the sources of nondeterminism during app loading, so we could not even save the execution state of the first script. On the other hand, Enyo framework does not call sources of nondeterminism source, so execution state of some global codes is included in the snapshot.

Deterministic DOM accelerate app loading by 1.38 times on average. It saves most execution results of framework and app script tags (global code) in the snapshot. Most enyo apps and some jQuery apps show the same performance improvement as *nondeterministic JS*, which ignores nondeterminism and saves the result of all event handlers.

Semi-randomized DOM accelerate app loading by 1.81 times on average. Although it still considers nondeterminism, it shows a performance improvement similar to *nondeterministic JS*, which accelerate app loading by 2.00 times. In most apps, *deterministic DOM* and *semi-randomized DOM* have the snapshot point at the same script, but the breakdom, photoshoot, and piratepig apps, which use lots of randomness, have a difference snapshot point. They use the *Math.random* function to calculate the location or the type of the DOM elements, which take a considerable amount of execution time. *Semi-randomized DOM* could achieve a substantial performance gain by including those executions in the snapshot.

7.2 Size of Nondeterministic Sets

We measured number of statements executed in the app loading. Table 3 shows the result. The first column shows the app names. The second column shows number of executed statements at app loading. The third and fourth columns show number of nondeterminism source statement and statements *reachable* from them. The fifth and sixth column show number of DOM change statement and number of DOM changes *affected* by nondeterminism. We measure LOC of jQuery framework and app code separately. Since the size of the enyo framework varies depending on the module used in the app, we analyzed minimal enyo framework code and both framework and app code together for enyo apps. We collected and averaged all execution results in the test runs.

The results show that many statements are reachable from nondeterminism at app loading. Although the number of nondeterminism sourzce takes less than 1% from executed statements, one source of nondeterminism propagates to average 379 statements. jQuery framework has nondeterminism but does not affect DOM tree with nondeterminism. We checked the statements executed during app loading to ensure that the tool correctly analyzes information on the statement and DOM tree mutations affected by nondeterminism, even in complex apps.

8 RELATED WORK

Wprof [27] classifies dependencies between browser activities and identifies bottleneck activities that contribute to the page load time using critical path analysis. They show that computation is a significant factor that constitute for 35% of the page load time on the critical path and synchronous JavaScript evaluation make a major contribution to page load time since it blocks parsing. Nejati et al. [15] analyzed web page load time for a mobile browser, which has limited hardward and poor network connection. On the mobile

Table 3: Number of executed statements during app loading

Web app	LOC	# exec stmt	# nondeter source	# nondeter affect	# DOM change	# nondeter DOM change
jQuery	10315	4736	4	4	7	0
breakdom	193	274064	217	253195	861	858
centnote	134	5457	5.4	625	7.6	0.6
cubeout	2180	77439	5	7	35	0
photoshoot	182	16384	25	14127	88	81
planner	584	2481	49	1081	113	7
timer	101	24851	7	15199	68	52
Enyo	4283	15153	0	0	0	0
bootplate	8730	36608	0	0	6	0
cryptotweets	8919	56296	29	253	28	0
piratepig	5068	80890	570	59392	14	2
Average	3037	58962	90.7	34388	122.1	100.1

browser, computation activities is the primary bottleneck which occupy more than 60% of the critical path.

There have been previous studies using snapshots to reduce app loading time. V8 JavaScript engine had a feature called *serialization*, which saves the JavaScript heap's built-in object in a file after the V8 initialization [5]. The V8 engine reads the serialized built-in object into the heap at startup, thus reducing the start-up time. Recently, using the APIs provided by V8, developers can also save the JavaScript execution results in the custom snapshot to accelerate the app loading. However, V8 can store only JavaScript state, and does not save the DOM state in the snapshot. There are earlier researches to accelerate app loading using a snapshot for the WebKit browser [16, 17]. Like V8, they traverse the JavaScript heap to save objects as snapshot files. It can also save the state of the DOM tree to extend the scope of the snapshot. The idea is recording all DOM change logs and restoring the DOM tree state by replaying it [16]. Recently, a different approach has been proposed where the DOM state is saved in serialized HTML [29]. It also provides a method of taking a snapshot point at the event handler, while the previous works take a snapshot at the script tag or at the end of app loading only. All these studies did not consider how to apply the snapshot when there is nondeterminism, so the developer had to analyze the app code and apply the snapshot on his own. This paper proposes a tool that can notify a proper snapshot point with two new nondeterminism-aware snapshot points.

Midas considered nondeterminism in middleware replication environments [22]. It has exploited application-level insight to classify nondeterminism as either affecting the application's persistent state or not. They used dynamic analysis to examine the effect of nondeterminism on the app and suggest ways to make the states the same when nondeterminism occurs in several devices. They classify nondeterminism based on the global state in the COBRA environment, and we categorize nondeterminism from the DOM tree in the web environment. We proposed a method to apply snapshot considering nondeterminism and made it easy to use snapshot through the tool.

Mugshot have developed record-and-replay for debugging for web environment and dealing with nondeterminism [14]. Mugshot classify the nondeterminism that can occur in the browser into five categories. *Tardis* also implemented record-and-replay in a managed runtime environment and considered nondeterminism [2]. They considered various nondeterminism such as network I/O, thread switches, etc. that can occur in the managed runtime, but focused on demonstrating the same execution result as the mugshot. We also classify nondeterminism that can occur in the web environment, but our research has focused on how to consider nondeterminism in environments using snapshots. In an environment where a snapshot is used, nondeterminism must be performed each time differently.

There have been various studies analyzing JavaScript. TypeDevil is research to find out when the type of variables changes dynamically through dynamic analysis [19]. They recorded all the information about how the type changed for each variable and analyzed it. ConflictJS is a study that tells you when a conflict occurs using the same variable between JavaScript libraries [18]. They analyzed the global variables that each JavaScript library writes to find conflicts between the libraries. Our study has analyzed nondeterminism occurring in a JavaScript environment differently from other studies. We also provide information about the scripts that can be included in the snapshot that developers can use based on the analysis.

9 CONCLUSION

In this paper, we analyzed JavaScript executions during app loading and found that nondeterminism occurs frequently but seldom affects the user's display. Based on these observations, we proposed two new snapshot points considering nondeterminism. We also developed a tool to inform the snapshot points automatically. We use the snapshot points obtained by the nondeterminism analysis tool for real apps and acclerate app loading by 1.81 times on average, compatitive to that of a snapshot point that completely ignores the nondeterminism issue.

ACKNOWLEDGMENTS

This work was supported by Basic Science Research Program through the National Research Foundation (NRF) of Korea funded by the Ministry of Science, ICT & Future Planning (NRF-2017R1A2B2005562).

REFERENCES

[1] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A Framework for Automated Testing of JavaScript Web Applications. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ACM Press, New York, USA, 571–580. https://doi.org/10.1145/1985793.1985871

[2] Earl T Barr and Mark Marron. 2014. Tardis: Affordable Time-travel Debugging in Managed Runtimes. *ACM SIGPLAN Notices* 49, 10 (2014), 67–82. https://doi.org/10.1145/2660193.2660209

[3] Tammy Everts. 2018. New findings: For top ecommerce sites, mobile web performance is wildly inconsistent. https://blog.radware.com/applicationdelivery/wpo/2014/10/2014-mobile-ecommerce-page-speed-web-performance/

[4] Dennis F Galletta, Raymond M Henry, Scott Mccoy, and Peter Polak. 2002. Web site delays: How tolerant are users? *Information Systems Research* 17, December 2002 (2002), 20–37.

[5] Google. 2018. Custom startup snapshots. https://v8.dev/blog/custom-startup-snapshots

[6] JQuery. 2018. jQuery. https://jquery.com/

[7] LG. 2018. Enyo JavaScript Application Framework. http://enyojs.com/

[8] LG. 2018. webOS. http://webosose.org/

[9] Xuanhua Li and Donald Yeung. 2007. Application-level correctness and its impact on fault tolerance. In *Proceedings - International Symposium on High-Performance Computer Architecture*. 181–192. https://doi.org/10.1109/HPCA.2007.346196

[10] Jim Liddle. 2008. Amazon found every 100ms of latency cost them 1% in sales. https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/

[11] Linux Foundation. 2012. Tizen. https://www.tizen.org/

[12] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node.js JavaScript applications. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015* (2015), 505–519. https://doi.org/10.1145/2814270.2814272

[13] David Sawyer McFarland. 2011. *JavaScript & jQuery: The Missing Manual*. O'Reilly. 538 pages.

[14] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot : Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. 159–174.

[15] Javad Nejati and Aruna Balasubramanian. 2016. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web - WWW '16*. ACM Press, New York, USA, 1305–1315. https://doi.org/10.1145/2872427.2883014

[16] JinSeok Oh. 2016. *Exploiting Snapshot for Web Applications*. PhD diss. Seoul National University. http://s-space.snu.ac.kr/handle/10371/119192

[17] JinSeok Oh and Soo Mook Moon. 2015. Snapshot-based loading-time acceleration for web applications. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*. IEEE, 179–189. https://doi.org/10.1109/CGO.2015.7054198

[18] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. ConflictJS: Finding and Understanding Conflicts Between JavaScript Libraries. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 741–751. https://doi.org/10.1145/3180155.3180184

[19] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering*, Vol. 1. IEEE Press, 314–324. https://doi.org/10.1109/ICSE.2015.51

[20] Selenium. 2018. Selenium, Browser Automation. https://www.seleniumhq.org/

[21] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM, 488–498. https://doi.org/10.1145/2491411.2491447

[22] Joseph Slember and Priya Narasimhan. 2006. Living with Nondeterminism in Replicated Middleware Applications. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., 81–100. http://dx.doi.org/10.1007/11925071_5

[23] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM event dependency analysis for testing web applications. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* (2016), 447–459. https://doi.org/10.1145/2950290.2950292

[24] Anne van Kesteren. 2015. MutationObserver. https://www.w3.org/TR/dom/#interface-mutationobserver

[25] Ted Vrountas. 2018. How Slow Mobile Page Speeds Are Ruining Your Conversion Rates. https://instapage.com/blog/optimizing-mobile-page-speed

[26] W3Techs. 2018. Usage of JavaScript libraries for websites. https://w3techs.com/technologies/overview/javascript_library/all

[27] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying page load performance with WProf. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation - NSDI'13*. 473–485.

[28] Jiabin Ye, Cheng Zhang, Lei Ma, Haibo Yu, and Jianjun Zhao. 2016. Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 449–459. https://doi.org/10.1109/SANER.2016.96

[29] JiHwan Yeo, JinSeok Oh, and Soo-Mook Moon. 2019. Accelerating Web Application Loading with Snapshot of Event and DOM Handling. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19)*.