Parallel JavaScript Execution in Web Navigation Sequences

Jose Losada
University of A Coruña
Campus Elviña s/n
A Coruña, Spain
jlosada@udc.es

Juan Raposo
University of A Coruña
Campus Elviña s/n
A Coruña, Spain
jrs@udc.es

Alberto Pan
University of A Coruña
Campus Elviña s/n
A Coruña, Spain
apan@udc.es

Paula Montoto
University of A Coruña
Campus Elviña s/n
A Coruña, Spain
pmontoto@udc.es

Manuel Álvarez
University of A Coruña
Campus Elviña s/n
A Coruña, Spain
mad@udc.es

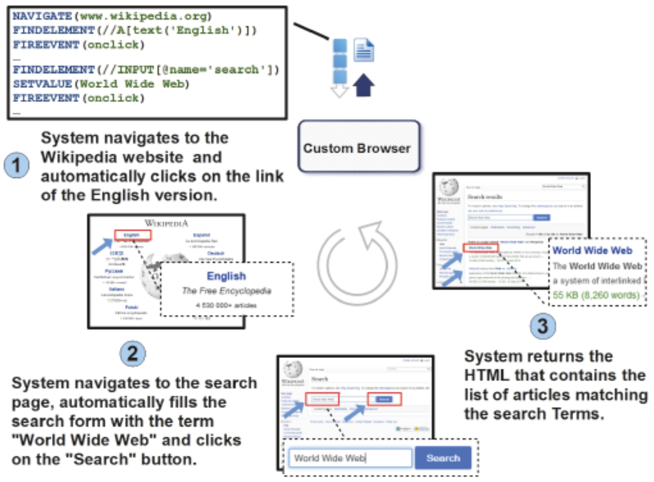
***Abstract*—Web automation applications are widely used for different purposes such as B2B integration and automated testing of web applications. Most current systems build the automatic web navigation component by using the APIs of conventional browsers. This approach suffers performance problems for intensive web automation tasks which require real time responses. Other systems use the approach of creating custom browsers specially designed for web automation. Those browsers can develop some improvements based in the peculiarities of the web automation tasks. In this paper, we present a novel optimization technique that allows the parallel execution of the JavaScript while the navigation component loads the web page. This technique is based in the analysis of the interactions between the scripts during the first loading of the web page, generating some useful information that will be saved and used in the next executions. The tests executed with real web sources show that the scripts contained in the HTML documents can be evaluated concurrently and the navigation component loads the web pages faster when the scripts are executed in parallel.**

Keywords*—*Web Automation, Navigation Sequence, Parallel JavaScript, Efficient Execution.

I. Introduction

Most today's web sources do not provide suitable interfaces for software programs. That is why a growing interest has arisen in so-called web automation applications. These applications are able to automatically navigate through websites, simulating the behavior of a human user and they are widely used for different purposes such as B2B integration, web mashups, automated testing of web applications, Internet meta-search or business and technology watch. For example, a technology watch application can use web automation to automatically search in the different websites and daily retrieve new patents and articles of a predefined area of knowledge.

A crucial part of web automation technologies is the ability to execute automatic web navigation sequences. An automatic web navigation sequence consists in a sequence of steps representing the actions to be performed by a human user over a web browser to reach a target web page. Figure 1 illustrates an example of a web navigation sequence that retrieves the list of articles matching the search term "*World Wide Web*" in the English version of the Wikipedia website.



```
NAVIGATE(www.wikipedia.org)
FINDELEMENT(//A[text('English')])
FIREEVENT(onclick)
…
FINDELEMENT(//INPUT[@name='search'])
SETVALUE(World Wide Web)
FIREEVENT(onclick)
…
```

Custom Browser
1
System navigates to the Wikipedia website and automatically clicks on the link of the English version.
English
The Free Encyclopedia
2
System navigates to the search page, automatically fills the search form with the term "World Wide Web" and clicks on the "Search" button.
World Wide Web
Search
3
System returns the HTML that contains the list of articles matching the search Terms.
World Wide Web
The World Wide Web
a system of interlinked
55 KB (8,260 words)

Fig. 1. Navigation Sequence Example

The approach followed by most of the current web automation systems [1], [2], [3], [4], [5] consists in using the APIs of conventional web browsers to automate the execution of navigation sequences. This approach does not require to develop a custom navigation component, and guarantees that the accessed web pages will behave the same as when they are accessed by a regular user. While this approach is adequate to some web automation applications, it presents performance problems for intensive web automation tasks which require real time responses. This is because conventional web browsers are designed to be client-side applications and they consume a significant amount of resources.

There exist other systems which use the approach of creating custom browsers to execute web navigation sequences [6], [7], [8]. Since they are not oriented to be used by humans, they can avoid some of the tasks of conventional browsers (e.g. page rendering). Nevertheless, they work like conventional browsers when loading and building the internal representation of the web pages.

We address this problem by using a custom browser specially built for web automation tasks. This browser is able to improve the response times and save a significant amount of resources because it supports a set of optimizations algorithms [9] not included in any other automation system.

Most of today's web pages use scripting languages intensively, mainly JavaScript, to increase its interactivity level and improve the site navigation. Therefore, the JavaScript execution is an important step in the loading process of the pages. In this work, we present a new set of techniques and algorithms that allow the parallel execution of the JavaScript code, while the custom browser loads a web page.

With these techniques, in a first execution of a navigation sequence, the navigation component will be able to detect the interactions between the scripts contained in the HTML document, saving some relevant information that will be used in the next executions of the same navigation sequence. Using this information, the custom browser will be able to execute in parallel the scripts without dependencies between them.

The rest of the paper is organized as follows: section II briefly describes the models our approach relies on. Section III presents an overview of the architecture and functioning of the conventional and custom browsers. Section IV explains the designed technique in detail and Section V explains its limitations. Section VI describes the experimental evaluation of the approach. Section VI. Section VII discusses related work. Finally, section VIII summarizes our conclusions.

II. Background

A. *Document Object Model*

The main model we rely on is the Document Object Model (DOM) [10]. This model describes how browsers internally represent the HTML web page currently loaded and how they respond to user actions.

An HTML page is modelled as a tree, where each HTML element is represented by an appropriate type of node. An important type of nodes are the script nodes, used to place and execute scripting code, typically written in some scripting language such as JavaScript.

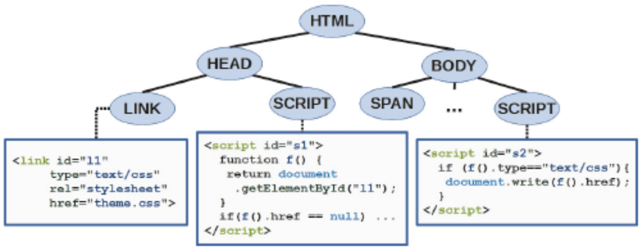
In addition, every node can receive events produced (directly or indirectly) by the user actions. Event types exist for actions such as clicking on an element (*click*), or moving the mouse over it (*mouseover*), to name but a few. Each node can register a set of listeners for the different types of events. An event listener has the entire page DOM tree accessible and can execute arbitrary scripting code.

B. *Dependencies Between nodes*

In our previous work [9], we introduced the concept of dependency between nodes of the DOM tree. This is a key concept in the design of the proposed optimization. We can summarize the idea with the following definitions:

Definition 1. We say that the node *n1* depends on the node *n2* when *n2* is necessary for the correct execution of *n1*. We say that *n2* is a dependency of *n1* and denote it as *n1*→*n2*. The following rules define this type of dependencies:

1. If the script code of a node *s1* uses an element, (e.g. a function or a variable) declared or modified in a previous script node *s2*, then *s1*→*s2*. Rationale: to be able to execute the script code of the node *s1* the node *s2* must be executed previously.



```
<link id="l1"
      type="text/css"
      rel="stylesheet"
      href="theme.css">
```

```
<script id="s1">
  function f() {
   return document
    .getElementById("l1");
  }
  if(f().href == null) ...
</script>
```

```
<script id="s2">
  if (f().type=="text/css"){
   document.write(f().href);
  }
</script>
```

Fig. 2. Example of Dependencies Between Nodes

2. If the script code of a node *s* uses a node *n*, then *s*→*n*. Rationale: to be able to execute the script code of the node *s*, the node *n* must be loaded previously. For instance, if *s* obtains a reference to an anchor node (e.g. using the function *getElementById* of the *document* object) and navigates to the URL specified by its *href* attribute, then it will not be possible to execute *s* unless the anchor node is present in the DOM.

3. If the script code of a node *s* makes a modification in a node *n*, then *n*→*s*. Rationale: the action performed by *s* may be needed to allow *n* to be used later. For instance, if *s* modifies the *action* attribute of a form node to set the target URL, then it will not be possible to submit the form unless *s* is executed previously.

Definition 2. We say that there exists a dependency conditioned to the event *e* being fired over the node *n*, between two nodes *n1* and *n2*, when the node *n2* is necessary for the correct execution of the node *n1*, when the event *e* is fired over the node *n*. We denote this as $n1 \xrightarrow{e|n} n2$. For instance, suppose *n* is a node with a listener for the *onMouseOver* event. The script code of the listener invokes a function defined in the script *s*. Then $n \xrightarrow{onMouseOver|n} s$.

Analogous rules to those explained previously define this type of dependencies, which, in this case, involve nodes containing event listeners.

Figure 2 illustrates an example of dependencies between nodes. The script with id *s1* defines and uses the function *f*. This function access the link node with id *l1* (using the function *getElementById*), so *s1*→*l1*. The script with id *s2* uses the function *f*, so *s2*→*s1*.

III. Overview

This section presents an overview about the architecture and functioning of the conventional browsers (section III.A) and custom browsers (section III.B).

A. *Conventional Web Browsers*

A web browser is a software application used for retrieving and presenting resources downloaded from the WWW.

Today's most popular web browsers are Mozilla Firefox [11], Google Chrome [12], Microsoft Internet Explorer [13], Apple Safari [14] and Opera [15].

1) *High Level Architecture*

The architecture of the modern web browsers [16] includes the high level components: Graphical User Interface, Browser Engine and Rendering Engine; and the auxiliary subsystems:

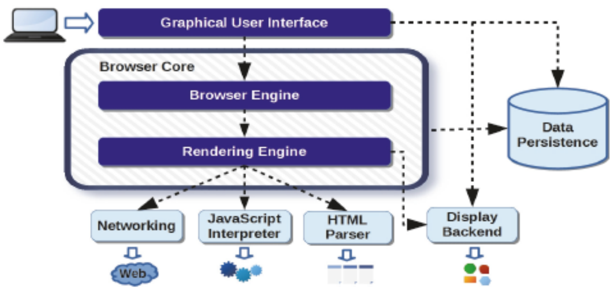


Fig. 3. Reference Architecture of Modern Web Browsers

JavaScript Interpreter, Networking, Display Backend, HTML parser and Data Persistence Layer. Figure 3 illustrates this architecture.

The Graphical User Interface includes the browser display area except the main window where the HTML page is rendered (address bar, toolbars, main menu, etc.).

The Browser Engine provides a high level interface for querying and manipulating the Rendering Engine. This component provides methods for high level browser actions, e.g., loading a URL, go back to the previous page, etc.

The Rendering Engine is responsible for parsing and displaying the HTML contents

2) Rendering Engine

The Rendering Engine represents the core of the web browser because it is responsible for parsing and displaying the HTML contents. This process fires a set of events in cascade and most of them are processed sequentially. The steps are illustrated in the Figure 4:

Download and Decode: HTML contents are downloaded from the web server and decompressed.

Processing: the DOM tree is built. For efficiency purposes, this is usually an incremental process. When new resources are discovered, they are downloaded (in parallel) and processed (e.g., images, style sheets, scripts, etc.). Style sheets contain presentation information that will be used to build the page layout. Each style sheet contains a list of rules that are applied on DOM nodes. Script nodes contain evaluable scripting code. During the script execution, the page layout can be dynamically recalculated.

Layout and Rendering: the layout tree contains rectangles with visual attributes (e.g. dimensions and colors). The rendering process paints the layout on the browser window using the display backend layer.

3) JavaScript Execution

Due to the semantics of JavaScript, web browsers execute the scripts in a sequential form. Nevertheless, there are some special cases where they can execute JavaScript code in parallel.

First, the scripts containing the attribute *async* can be executed asynchronously with the rest of the page loading. This feature has been introduced in HTML5 [17], and is available only for external scripts (should only be used if the *src* attribute is present).

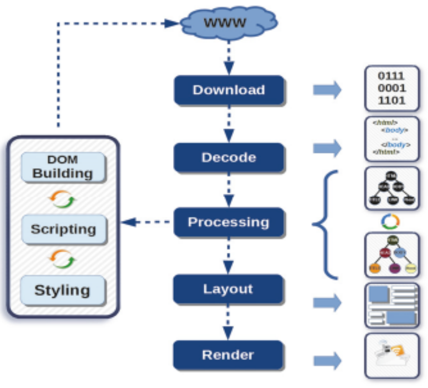


Fig. 4. Rendering Engine Processing Steps

The other scenario where the JavaScript can be executed in parallel is using Web Workers (also introduced in HTML5). A Web Worker can execute JavaScript in background but have the major limitation that the scripting code cannot access the DOM tree objects.

Figure 5 shows a real example of the browser Google Chrome loading the web page *github.com*. It shows the Timeline View included in the Developer Tools, and it displays the events fired in the rendering engine of the browser, while the page is being loaded.

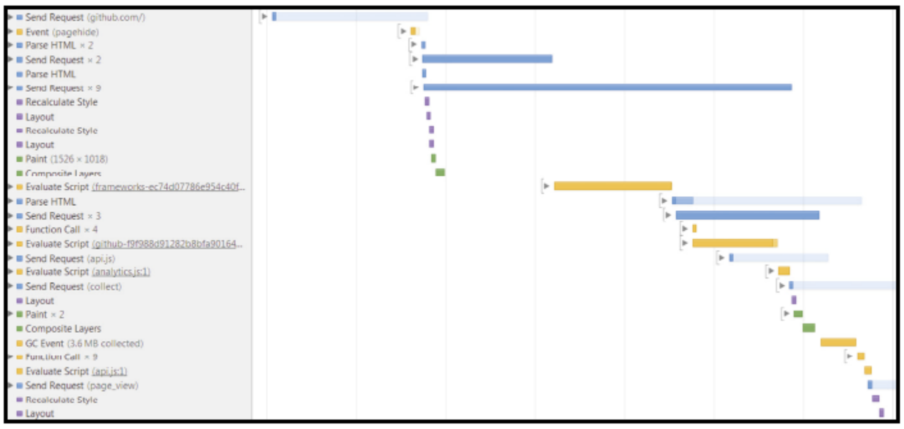
Network calls are represented with *Send Request* actions, HTML processing events are represented with *Parse HTML* actions and JavaScript executions are represented with *Evaluate Script* and *Function Call*. The web page contains several style sheets, images and scripts that are downloaded in parallel at the beginning (*Send Request x 2* and *Send Request x 9* events). Mostly, the script evaluation is executed sequentially but there is some level of parallelism because some scripts are marked with the attribute *async* (*Function Call x 4* is executed in parallel with the evaluation of *github.js*). *Function Call x 4* and *Function Call x 9* represent, mostly, the execution of functions invoked with *SetTimeout* (during the evaluation of *frameworks.js*). There is a fourth script in the page (*api.js*) dynamically generated during the evaluation of *github.js*. The rendering and painting events are fired during the entire page loading.

B. *Custom Web Browsers*

Custom browsers are navigation components used in web automation systems, specialized in the execution of navigation sequences. Figure 6 illustrates its architecture.

Custom browsers usually simulate the behavior of a real browser and they are designed with two main goals: the perfect emulation of a conventional browser (if the custom browser does not behave just the same as a real browser, the sequence execution could lead to wrong web pages) and the efficiency in the execution of the navigation sequences.

Custom browsers are not human-oriented and the visualization of the results is not necessary. This will increase the efficiency because there is no need for building and render the page layout.



```
Send Request (github.com/)
Event (pagehide)
Parse HTML × 2
Send Request × 2
Parse HTML
Send Request × 9
Recalculate Style
Layout
Recalculate Style
Layout
Paint (1526 × 1018)
Composite Layers
Evaluate Script (frameworks-ec74d07786e954c40f...
Parse HTML
Send Request × 3
Function Call × 4
Evaluate Script (github-f9f988d91282b8bfa90164...
Send Request (api.js)
Evaluate Script (analytics.js:1)
Send Request (collect)
Layout
Paint × 2
Composite Layers
GC Event (3.6 MB collected)
Function Call × 9
Evaluate Script (api.js:1)
Send Request (page_view)
Recalculate Style
Layout
```

Fig. 5. Google Chrome Timeline

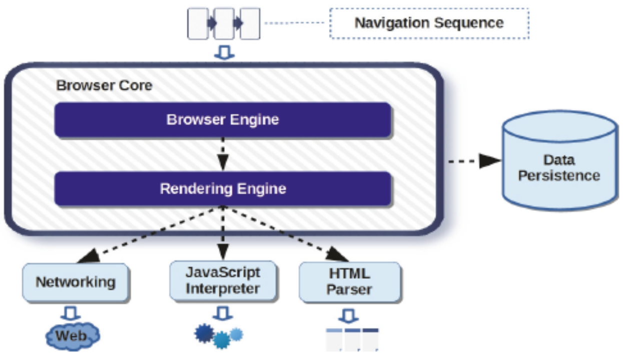


Fig. 6. Custom Browser Architecture

In the custom browsers architecture, the Browser Engine is also the entry point for accessing the Rendering Engine, but it does not receive commands from the user interface. Instead, the Browser Engine receives the list of commands of the navigation sequence to be executed. These commands will represent events produced by a human user in a real web browser, e.g., navigations to URLs or events over the DOM elements of the loaded page.

In the web automation systems, where custom browsers are used, navigation sequences are known 'a priori' and executed multiple times.

This peculiarity can be used to extract some useful information during a first execution of each navigation sequence (at definition time) with the goal to use it in the following executions and improve the efficiency. The optimization proposed in this paper exploits this peculiarity, and the script elements identified as not dependent (in a first execution) are executed in parallel (in the next executions) without affecting the correct loading of the page.

IV. Parallel JavaScript Execution

In this section, we explain in detail the proposed technique to execute JavaScript in parallel. In our approach, the custom browser works in two phases: optimization and execution. The same approach was used in [9] to design an optimization technique related with the detection of the irrelevant fragments of a document, not required for the correct execution of the navigation sequence.

Algorithm: builds the scripts dependency ghaph.
Inputs: 1. *dependencyGraph:* graph with the dependencies between DOM nodes.
2. *DOM:* DOM tree of the document.
Output: *scriptDependencyGraph* with the dependencies between script nodes.

```
function BuildScriptsDependencyGraph(dependencyGraph,DOM) {
   var scriptDependencyGraph ← new Graph()
   for each node in DOM {
     if (node.isScript()) {
       AddScriptDependencies(node,dependencyGraph,node,scriptDependencyGraph);
       if(not scriptDependencyGraph.Contains(node)) {
         scriptDependencyGraph.AddSafeScript(node)
       }
     }
   }
   return scriptDependencyGraph
}
```

Algorithm: recursivelly collect *script* dependencies starting in the *node* dependencies.
Inputs: 1. *node*: node to check if contains *script* dependencies
2. *dependencyGraph*: graph with the dependencies between DOM nodes.
3. *script*: script node to add dependencies.
4. *scriptDependencyGraph*: i/o parameter with the dependencies between scripts.
Output: *script* dependencies will be added to *scriptDependencyGraph*.

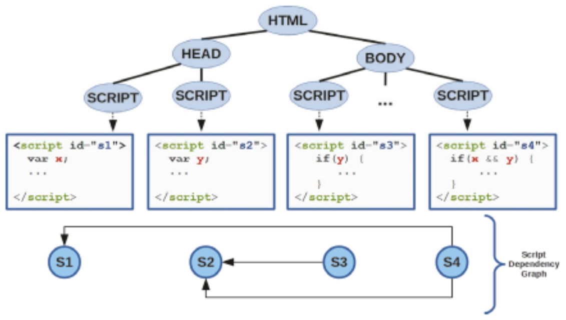
```
function AddScriptDependencies(node,dependencyGraph,script,scriptDependencyGraph) {
  var nodeDependencies ← dependencyGraph.Get(node);
  for each nodeDependency in nodeDependencies {
    if (nodeDependency.isScript()) {
      scriptDependencyGraph.Add(script, nodeDependency)
    }
    AddScriptDependencies(nodeDependency,dependencyGraph,script,
                          scriptDependencyGraph)
  }
}
```

Alg. 1. Script Dependency Graph

It can be summarized as follows: in the optimization phase, the navigation sequence is executed once and, in the meantime, the navigation component automatically calculates some optimization information and saves it. In the execution phase, the rendering engine uses the information stored during the optimization phase to execute the same navigation sequence efficiently. In the following paragraphs, we explain the actions executed by the navigation component in each phase.

As commented previously, in the optimization phase, the navigation sequence is executed once. During this execution, the script evaluation is monitored to collect dependencies between the nodes in the DOM tree. Dependencies are calculated according to the rules detailed in the section II.B. At this point, a dependency graph is generated. This graph stores for each node in the DOM, the other nodes that are dependencies of it. This part of the optimization phase is common to the technique presented in [9].

After generating the dependency graph, it is used to build another structure containing only the dependencies between scripts. This structure is named script dependency graph and it will include, for each script S in the page, the list of other scripts that must be executed before.



```
<script id="s1">
  var x;
  ...

</script>
```

```
<script id="s2">
  var y;
  ...

</script>
```

```
<script id="s3">
   if(y) {
      ...
   }
</script>
```

```
<script id="s4">
   if(x && y) {
      ...
   }
</script>
```

Fig. 7. Script Dependency Graph Example

If these scripts, dependencies of *S*, are not executed before, the evaluation of the script *S* will fail (e.g. these scripts define variables or functions used in the script *S*).

Algorithm 1 describes the pseudo-code for the generation of the script dependency graph. The inputs of the main function (*BuildScriptsDependencyGraph*) are: the graph with the dependencies between all nodes in the DOM tree (*dependencyGraph*) and the DOM tree. The algorithm iterates over each node in the tree, looking for scripts. For each script, the dependencies are generated searching recursively in the dependency graph for other scripts (*AddScriptDependencies* function).

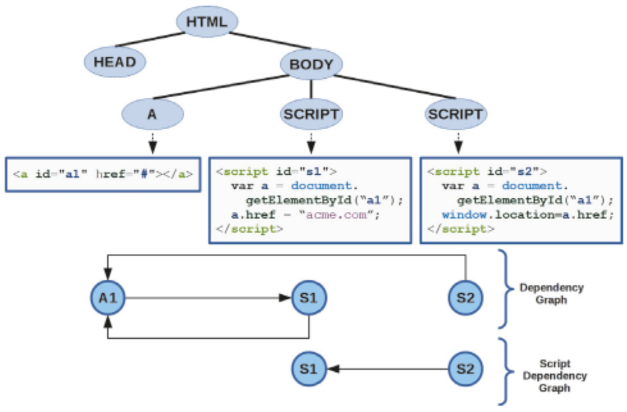
For optimization purposes, scripts without dependencies with other scripts, are saved in a special list of safe scripts (line 7).

Figure 7 illustrates an example of script dependency graph generation. The example shows a DOM tree with four script nodes.

The first script identified with the id attribute *s1*, defines the variable *x*. This script does not have dependencies with other scripts, so it will be added to the list of safe scripts. The second script, *s2*, defines the variable *y*. This script does not have dependencies with other scripts either, so it is also a safe script and its execution can be parallelized with *s1*.

The third script, *s3*, uses the variable *y* defined in *s2*. The script dependency graph will include a dependency between *s3* and *s2*. This means that *s3* will not be executed until *s2* finishes its execution, but *s3* can be parallelized with *s1* because there are no dependencies between them. The four script, *s4*, uses the variable *y* defined in *s2* and also the variable *x* defined in *s1*. The script dependency graph will include *s1* and *s2* in the dependencies of *s4*. This means that *s4* will not be executed until *s1* and *s2* finish its execution but *s4* can be parallelized with *s3* because there are no dependencies between them.

Figure 8 shows another example of script dependency graph generation. In this case, the script execution involves not only scripts but also another DOM nodes. This example includes two scripts and one anchor node, *a1*. The first script, *s1*, uses the anchor node *a1*, (accessed with the function *getElementById*).



```
<a id="a1" href="#"></a>
```

```
<script id="s1">
  var a = document.
    getElementById("a1");
  a.href = "acme.com";
</script>
```

```
<script id="s2">
  var a = document.
    getElementById("a1");
  window.location=a.href;
</script>
```

Fig. 8. Script Dependency Graph Example with Nodes

Algorithm: tests if a script is ready to run.
Inputs: 1. *script*: script node to test if it is ready to run.
2. *scriptDependencyGraph*: graph with dependencies between scripts.
Output: *true* if the script can be executed.

```
 1:  function IsReadyForExecution(script, scriptDependencyGraph){
 2:      if(scriptDependencyGraph.IsSafeScript(script)) {
 3:          return true
 4:      }
 5:      var dependencies ← scriptDependencyGraph.Get(script);
 6:      for each scriptDependency in dependencies {
 7:          if (not scriptDependency.HasFinished()){
 8:              return false
 9:          }
10:      }
11:      return true
12:  }
```

Alg. 2. Script Ready for Execution

Applying the rules defined in the section 2, *s1*→*a1*. But the script *s1* also modifies the *href* attribute of the anchor node, then *a1*→*s1*. The second script, *s2*, also uses the anchor node *a1*, then *s2*→*a1*. In addition, *s2* uses the *href* attribute of the anchor node, previously modified during the execution of *s1*. So, in this case, for the correct execution of *s2*, *s1* must be executed previously and their execution cannot be parallelized. This is reflected in the script dependency graph, including *s1* as dependency of *s2*.

Once the script dependency graph is calculated we need to generate expressions to be able to identify the script nodes contained in it, during the execution phase. For that purpose we use XPath-like [18] expressions generated using a previously designed algorithm explained in detail in [9].

During the execution phase (involves the subsequent executions of the same navigation sequence), the script dependency graph is used to identify the scripts that can be executed in parallel.

Algorithm 2 describes the pseudo-code of the algorithm that tests if a script is ready for execution. A script is considered ready for execution when it has no dependencies (*IsSafeScript*, line 2) or when all its dependencies have been already executed and have finished (*HasFinished*, line 7). When a script finishes its execution, the dependency graph is updated, marking that script as executed.

Algorithm 3 describes how the browser events, generated during the execution of the navigation sequence, are dispatched (including the script events that can be executed in parallel).

Algorithm: executes one pending event in the input *thread*.
Inputs: 1. *thread*: idle thread ready to execute pending events.
2. *eventQueue*: queue containing pending events.
3. *scriptDependencyGraph*: graph with script dependencies.
Output: pending event is dispatched in the input *thread*.

```
 1: function execute(thread, eventQueue, scriptDependencyGraph){
 2:   if (eventQueue.IsEmpty()) return
 3:   if (thread.IsMainThread()) {
 4:     var event ← eventQueue.Pop()
 5:     if (scriptDependencyGraph is null or not event.IsScriptEvent()) {
 6:        thread.dispatch(event)
 7:        return
 8:     }
 9:     var script ← event.GetScript()
10:     if (script.IsAsync() or
11:        IsReadyForExecution(script, scriptDependencyGraph) ){
12:        thread.dispatch(event)
13:     }
14:     return
15:   }
16:   for each event in eventQueue {
17:     if (not event.IsScriptEvent()) {
18:        continue
19:     }
20:     var script ← event.GetScript()
21:     if (script.IsAsync() or
22:        IsReadyForExecution(script, scriptDependencyGraph) ){
23:        thread.dispatch(event)
24:        return
25:     }
26:   }
27: }
```

Alg. 3. Event Dispatching Process

This algorithm is designed for a custom browser that includes the following components in its Rendering Engine:

1) Event queue: stores the browser events for its execution. This queue will include different kind of events: HTML parsing, script execution, user action handlers, etc. The execution of events can produce new events that are also placed in the queue.

2) Main thread: executes browser events of any kind (e.g. script execution, HTML parsing, etc.). If the navigation sequence is executed without using the parallel script execution technique, all events are executed sequentially in this thread, emulating the behavior of the conventional browsers.

3) Pool of threads: executes only script events in parallel. This pool is used only if the script dependency graph is available for the web page that is being loaded.

The event dispatching algorithm receives three arguments: the idle thread that will execute one pending event, the event queue that stores the events pending for its execution and the script dependency graph with the information about the dependencies between the scripts.

First, the execution algorithm checks if the event queue contains pending events (*IsEmpty*, line 2). If the event queue is empty, the algorithm stops. Then, the algorithm checks the calling thread. If this thread is the main execution thread (*IsMainThread*, line 3), it will try to execute the first event in the queue (obtained using the function *Pop*, line 4). If this event is a script event (*IsScriptEvent*, line 5), the algorithm checks if that script is ready to be executed. The script is ready to be executed in two scenarios: if it contains the attribute *async* (*IsAsync*, line 10) or when it doesn't have unfinished dependencies (*IsReadyForExecution*, line 11; described in the Algorithm 2). Remember that the main thread can execute any type of events, including script events. When this thread finishes the execution of the event, it will call the execution algorithm again to select and execute the next event in the queue.

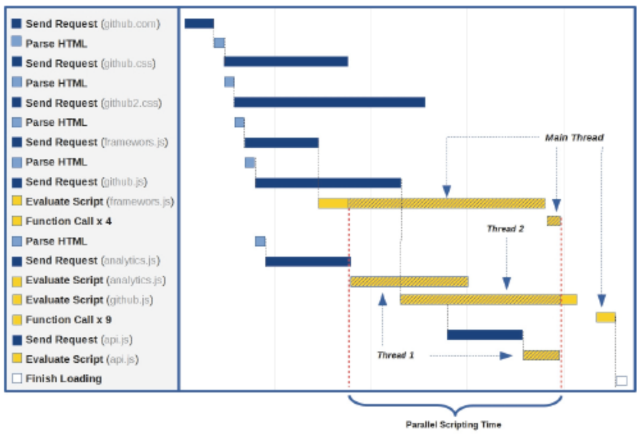


Fig. 9. Custom Browser Execution Timeline

If the calling thread is a thread of the pool, the algorithm iterates through the events in the queue, looking for scripts events (*IsScriptEvent*, line 17). Remember that the threads of the pool can only execute script events. When the algorithm finds a script in the queue, it checks if that script is ready to be executed.

When a script finishes its execution, the dependency graph is updated to mark that script as executed, so other scripts depending on it could switch to the ready state. Then, the algorithm is invoked again to identify and execute pending scripts that are ready to be executed.

Note that, in our model, the script events can be executed in parallel with any other type of events, including the HTML parsing events, which parse the input HTML and build the DOM tree incrementally. Therefore, when a script event is being executed, elements that appear below it in the HTML source could have already been added, as nodes, to the DOM tree. This could affect to the execution of the scripts.

For example, in the Figure 2, the script *s1* should not access the *span* node because the *script* node is located before it in the tree. The script engine of a custom browser using our parallel dispatching algorithm should deal with this situation, and ensure that the execution of a script considers only the DOM nodes corresponding to elements that appear above the script in the HTML code.

This can be implemented, for example, assigning consecutive numeric identifiers to each DOM node, and limiting the scope of each script to the nodes with a lower identifier.

Figure 9 illustrates the timeline of the events generated during a page loading process using the parallel script execution technique. The example shows the same page loaded in Google Chrome in Figure 5 and assumes that the optimization information was already generated and it is available at the execution phase. The scripts in the page were identified as safe scripts completely parallelizable because they have no dependencies. Network calls are represented with *Send Request* actions, HTML processing events are represented with *Parse HTML* actions and JavaScript executions are represented with *Evaluate Script* and *Function Call* actions.

The page loading involves the downloading of CSS style sheets, three JavaScript files located in the HTML, and a fourth JavaScript file (*api.js*) generated dynamically during the evaluation of *github.js*. The HTML parser incrementally discovers the external objects and starts downloading the files in parallel. The scripts are also evaluated in parallel because they were identified as completely independent during the optimization phase (note that they even start their execution in a different order from Figure 5). The dashed lines in the JavaScript events, represent the parallelization time. *Function Call x 4* and *Function Call x 9* are events generated during the evaluation of *frameworks.js*. *Function Call x 9* has a little delay because it is started with the *setTimeout* function.

V. Known Limitations

In some scenarios, even though the algorithm used to identify the script nodes is resilient to small changes, when the page changes, some of them could not be identified in the new version of the web page.

If that occurs, a script with unidentified dependencies cannot be parallelized and must be executed after waiting for all the previous scripts of the page to finish its execution. If new scripts appear in the page, the script dependency graph is invalidated and it should be recalculated.

VI. Evaluation

To evaluate the validity of the proposed optimization technique, we implemented a custom browser that emulates Microsoft Internet Explorer. This navigation component is implemented in Java using open source libraries. In the experiments, we selected websites from different domains and different countries, included in the top 500 sites on the web according to Alexa [19]. We executed a navigation sequence that loads one page in each one of the selected web sites (the home page in most of the cases). The test machine was a quad-core processor with 16GB of RAM. The thread pool size (for parallel script evaluation) was limited to a maximum of 3.

We compared the script execution time (networking time and HTML parsing time are not included) of the custom browser executing the scripts in parallel (using the script dependency graph previously calculated during the optimization phase), with a regular execution of the custom browser evaluating the scripts sequentially (without parallelism). We performed 30 consecutive executions of the navigation, discarding the results that do not fit in the range of the standard deviation.

Table I includes the following columns: the first one shows the script execution time (in milliseconds) of the regular execution, the second one shows the script execution time (in milliseconds) of the execution using the parallel script optimization (it also shows between brackets the percentage of improvement). The average improvement executing scripts in parallel is 22% (it varies from 6% of improvement in the worst case to 48% in the best case). Discarding the results that do not fit in the range of the average ± standard deviation, the average improvement is 20% and the median value is also 20%.

We do not include a detailed analysis of the time consumed during the optimization phase to generate the optimization information (i.e. the node dependencies and the script dependency graphs) and the time consumed during the execution phase to use that information (i.e. to check if a script node can be executed), due to space restrictions. Some of the algorithms are the same as the ones used in our previous work [9] (e.g. calculating the node dependencies or the XPath-like expression to identify a node), and we have demonstrated, in that work, that these algorithms consume an insignificant time compared to the execution time of the navigation sequence. Additionally, the new algorithms introduced in this work have a complexity with the same or even lower order of magnitude (e.g. the time spent manipulating the script dependency graph at execution phase is always less than 1 millisecond).

TABLE I. Parallel Scripts

	SCRIPTING TIME (MS)	PARALLEL SCRIPTING TIME (MS)
360.CN	392	304 (23%)
ALIBABA.COM	97	81 (17%)
ALLEGRO.PL	425	348 (19%)
AMAZONWS.COM	319	234 (27%)
BBC.COM	203	144 (30%)
BET365.COM	195	116 (41%)
BILD.DE	1098	806 (27%)
BLOGGER.COM	99	65 (35%)
BLOGSPOT.COM	112	66 (42%)
BLOOMBERG.COM	632	529 (17%)
BOOKING.COM	518	438 (16%)
CNET.COM	235	174 (26%)
ENGADGET.COM	447	274 (39%)
FORBES.COM	432	390 (10%)
GITHUB.COM	330	237 (29%)
GIZMODO.COM	80	60 (25%)
GSMARENA.COM	390	335 (15%)
IGN.COM	1228	1036 (16%)
IKEA.COM	81	69 (15%)
IMGUR.COM	686	549 (20%)
INDIATIMES.COM	977	819 (17%)
INSTAGRAM.COM	261	169 (36%)
LEMONDE.FR	305	203 (34%)
LIBERO.IT	396	297 (25%)
LIFEHACKER.COM	86	66 (24%)
LINKEDIN.COM	185	146 (22%)
LIVE.COM	300	266 (12%)
LIVEJOURNAL.COM	106	92 (14%)
MARCA.COM	723	515 (29%)
MASHABLE.COM	145	101 (31%)
MEDIAFIRE.COM	455	357 (22%)
PETFLOW.COM	727	557 (24%)
PINTEREST.COM	210	173 (18%)
REDIFF.COM	290	245 (16%)
REUTERS.COM	831	725 (13%)
RT.COM	605	478 (21%)
SCRIPBD.COM	689	579 (16%)
SOFTONIC.COM	537	483 (11%)
SOURCEFORGE.NET	504	327 (36%)
SPEEDTEST.NET	770	688 (11%)
STACKEXCHANGE.COM	255	222 (13%)
TAOBAO.COM	191	153 (20%)
TARINGA.NET	1494	1405 (6%)
TECHCRUNCH.COM	249	197 (21%)
THEFREEDICTIONARY.COM	503	424 (16%)
TIME.COM	329	204 (38%)
TRIPADVISOR.COM	131	103 (22%)
TUMBLR.COM	546	418 (24%)
UPLOADED.NET	412	344 (17%)
UPS.COM	1167	1013 (14%)
USATODAY.COM	185	120 (36%)
WARRIORFORUM.COM	346	312 (10%)
WEATHER.COM	516	454 (13%)
WIX.COM	500	433 (14%)
WORDPRESS.COM	63	33 (48%)
WORDREFERENCE.COM	185	120 (36%)
XDA-DEVELOPERS.COM	689	566 (18%)
YAHOO.COM	166	94 (44%)
YOUTUBE.COM	325	298 (9%)
ZIPPYSHARE.COM	563	470 (17%)
AVERAGE		22%
AVERAGE ± STDEV		20%
MEDIAN		20%

VII. Related Work

Most of the current web automation systems (Smart Bookmarks [1], Kapow [2], WebMacros [4], Selenium [5], and Pan et al. [3]) use the APIs of conventional web browsers to automate the execution of navigation sequences. This approach has two important advantages: it does not require to develop a new browser (which is costly), and it is guaranteed that the page will behave in the same way as when a human user access it with her browser.

Nevertheless, this approach presents performance problems for intensive web automation tasks which require real time responses. This is because traditional web browsers are human-oriented client-side applications and they consume a significant amount of resources and time, as we have demonstrated in our previous work [9].

Other systems use the approach of creating simplified custom browsers. For example, Jaunt [8] lacks the ability to execute JavaScript.

HtmlUnit [7] and EnvJS [6] use their own custom browser with advanced JavaScript support. They are more efficient than conventional browsers, because they are not oriented to be used by humans and can avoid some tasks (e.g. page rendering). Nevertheless, they work like conventional browsers when building the internal representation of the web pages.

To the best of our knowledge, the conventional browsers (Firefox, Chrome, etc.) execute the JavaScript sequentially, except in the particular scenarios explained in the section III.A: scripts with the *async* attribute and Web Workers. This support is very limited because the *async* attribute must be manually specified in the appropriate script elements of the HTML source code, and Web Workers cannot access to the DOM tree of the page currently loaded. Nevertheless, conventional browsers implement other type of optimizations, for example, Mozilla Firefox uses the speculative parsing [20] to early discover new resources and initiate preload actions in parallel.

Other browsers exploit different levels of optimization and parallelism, e.g., ZOOMM [21] is a parallel browser that exploits HTML pre-scanning with resource prefetching, concurrent CSS styling and parallel script compilation, and Adrenaline [22] speed up page processing by splitting the original page in mini-pages, rendering each one of these mini-pages in a separate process. None of these browsers implements the optimization described in this paper.

VIII. Conclusions

In this paper we presented a novel optimization technique for a custom browser specialized in the execution of web navigation sequences. This technique consists in the parallel execution of the JavaScript code contained in HTML documents. The designed algorithms allow the detection of the scripts in the page that can be executed in parallel because there are no dependencies between them. Those scripts are evaluated concurrently when the custom browser builds the DOM trees of the accessed web pages.

Our approach is based on executing the navigation sequence once, and automatically collect some relevant information about the interactions and dependencies between all the nodes in the web page. With this information, a script dependency graph is built. The script dependency graph will contain, for each script *S* in the page, the list of other scripts that must be executed before for the correct execution of *S*. The script dependency graph is saved using XPath-like expressions to represent the script nodes and is used in the next executions of the same navigation sequence, to detect those scripts that can be executed concurrently using a pool of threads.

The experimental evaluation executed using real web sites, demonstrate that the scripts in the page can be executed in parallel because there are no dependencies between them, and also demonstrate that the navigation component loads the HTML pages faster using the parallel scripting technique.

References

[1] Hupp D., Miller R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, pp. 81–90. ACM New York, Newport (2007).

[2] Kapow. http://kapowsoftware.com.

[3] Pan A., Raposo J., Álvarez M., Hidalgo J., Viña A.: Semi automatic wrapper generation for commercial web sources. In: IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context, pp. 265–283. Kluwer, B.V. Deventer, Japan (2002).

[4] Safonov A., Konstan J., Carlis J.: Beyond hard to reach pages: interactive, parametric web macros. In: 7th Conference on Human Factors & the Web. Madison 2001.

[5] Selenium. http://seleniumhq.org.

[6] EnvJS. http://www.envjs.com.

[7] HtmlUnit. http://htmlunit.sourceforge.net.

[8] Jaunt. Java Web Scraping & Automation. http://jaunt-api.com.

[9] Losada J., Raposo J., Pan A., Montoto P.: Efficient execution of web navigation sequences. World Wide Web Journal. DOI 10.1007/s11280-013-0259-8. ISSN 1386-145X.

[10] Document Object Model (DOM). http://www.w3.org/DOM/

[11] Mozilla Firefox. https://www.mozilla.org/en-US/firefox/

[12] Google Chrome. http://www.google.com/chrome/

[13] Internet Explorer. http://windows.microsoft.com/internet-explorer/

[14] Apple Safari. http://www.apple.com/safari/

[15] Opera Browser. http://www.opera.com/

[16] Grosskurth, A., Godfrey, M. W., September 2005. A reference architecture for web browsers. In: ICSM'05: Proceedings of the 21st IEEE Int. Conf. on Software Maintenance (ICSM'05). pp. 661–664.

[17] HTML5. https://html.spec.whatwg.org.

[18] XML Path Language (XPath), http://www.w3.org/TR/xpath.

[19] Alexa. The Web Information Company. http://www.alexa.com.

[20] Mozilla HTML5 Parser. https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/HTML5_Parser.

[21] Calin Cascaval, Seth Fowler, Pablo Montesinos Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. 2013. ZOOMM: a parallel web browser engine for multicore mobile devices. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13). ACM, New York, NY, USA, 271-280.

[22] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 2012. USENIX Association.