Concurrent JavaScript Parsing for Faster Loading of Web Apps

HYUKWOO PARK, MYUNGSU CHA, and SOO-MOOK MOON, Seoul National University

JavaScript is a dynamic language mainly used as a client-side web script. Nowadays, web is evolving into an application platform with its *web apps*, and JavaScript increasingly undertakes complex computations and interactive user interfaces, requiring a high-performance JavaScript engine. There have been many optimizations for efficient JavaScript engines, but one component that has not been optimized much is *JavaScript parsing*. A JavaScript function needs to be parsed before being executed, and the parsing overhead takes a substantial portion of JavaScript execution time for web apps, especially during *app loading*. This article proposes *concurrent parsing* of JavaScript, which performs the parsing of JavaScript functions in advance on different threads, while the main thread is executing the parsed JavaScript functions. This can hide the parsing overhead from the main execution thread, reducing the JavaScript execution time, thus reducing the overall app loading time. More specifically, we separated JavaScript parsing and made it run on different threads without violating the execution semantics of JavaScript. We also designed an efficient multi-threaded parsing architecture, which reduces the synchronization overhead and schedules the parsing requests appropriately. Finally, we explored two methods of choosing the target functions for concurrent parsing: one based on profiled information and the other based on speculative heuristics. We performed experiments on the WebKit browser with the JSC engine for real web apps. The result shows that the proposed concurrent parsing can improve the JavaScript performance during app loading by as much as 64% and by 39.7% on average. This improves the whole app loading performance tangibly, by as much as 32.7% and by 18.2%, on average.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Software and its engineering** → *Parsers*; *Scripting languages*

Additional Key Words and Phrases: JavaScript, web app, parser, concurrent parsing, web browser, javascript engine

ACM Reference Format:
Hyukwoo Park, Myungsu Cha, and Soo-Mook Moon. 2016. Concurrent JavaScript parsing for faster loading of web apps. ACM Trans. Archit. Code Optim. 13, 4, Article 41 (November 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/3004281

1. INTRODUCTION

JavaScript is a standard programming language for web pages along with HTML and CSS. HTML expresses the web components, CSS controls the visual effects, and JavaScript makes the computations, particularly for interacting with the user input and for dynamically changing the web pages in an event-driven manner. So, the role of JavaScript was relatively light and its performance was not an urgent issue.

This work was supported by the Ministry of Trade, Industry & Energy (MOTIE) through the Electronics and Telecommunications Research Institute (ETRI) (Project No. 10045344), and by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (R0126-15-1100, Development of web-accelerated software framework for web-based mobile service ecosystem).

Authors' addresses: H. Park, M. Cha, and S.-M. Moon, Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, Republic of Korea; emails: {clover2123, mscha, smoon}@altair.snu.ac.kr.

DOI: http://dx.doi.org/10.1145/3004281

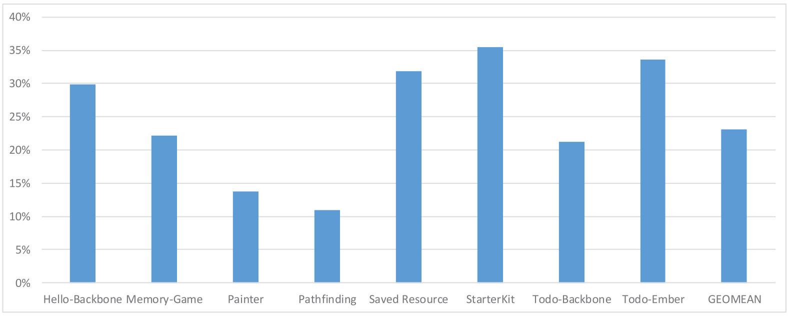


Fig. 1. JavaScript parsing portion in the loading time of web apps.

Since the appearance of new standards and technologies such as HTML5 [Hickson and Hyatt 2011], ECMAScript6 [ECMA International 2015], and WebGL [Marrin 2011], web has been rapidly transformed into a "complete" application platform such as Tizen [2016], webOS [LG, Inc. 2016], or Firefox OS [Mozilla 2016a]. For large-scale web applications (*web apps*) on these platforms, JavaScript plays an important role for performing complex computations or user interactivities. So, the JavaScript performance is important, especially considering its language features such as dynamic typing, first-class function, or closure, which are hard to execute efficiently.

Many optimizations have been made to improve the performance of the JavaScript engine, such as fast interpretation of JavaScript code [Pizlo 2014], just-in-time compilation (JITC) of JavaScript code to machine code [Gal et al. 2009], or fast initialization of JavaScript built-in objects [Thompson 2015]. Generally, modern JavaScript engines in WebKit [Apple, Inc. 2016a], Blink [Google. Inc. 2016a] and Firefox [Mozilla 2016b] include a multi-tier architecture based on adaptive compilation, which can generate high-performance code without heavy compilation overhead.

However, one component that has been optimized little, but is getting more important for web apps is *JavaScript parsing*, which we want to optimize in this article. In modern JavaScript engines, every JavaScript code to be executed should first go through the parsing process, which parses the source code and generates an intermediate representation, e.g., the bytecode in WebKit. Then, the interpreter executes the bytecode initially. When the JavaScript code is found to be hot, JITC translates the bytecode to machine code for better performance. When it is found to be really hot, the bytecode is compiled again by higher-tier JITC for high performance. We found that the parsing overhead takes a substantial portion of JavaScript execution time for web apps, especially during *app loading*. Figure 1 shows the JavaScript parsing overhead during the app loading time (from the start of app until the *load* event fires) for some JavaScript-heavy web apps on the WebKit browser. It is an average of 23% of the whole app loading time, which is surprising since app loading includes many other things such as HTML parsing, CSS rendering, layout, script download as well as JavaScript execution with interpretation, JITC, and GC.

Table I shows the number of JavaScript functions in each app: (a) shows the total number of functions and (b) shows the number of parsed (i.e., executed at least once) functions among them. Table I(c), (d), and (e) show the number of *cold*, *warm*, and *hot* functions classified by WebKit, meaning executed 1∼10 times, 10∼100 times, and more than 100 times, respectively; they are interpreted, compiled by baseline JITC, and compiled by the higher-tier JITC (DFG), respectively. Table I indicates that some functions are parsed, yet most of them are executed infrequently. We can imagine

Table I. Distributions of JavaScript Functions in Each App

	Hello Backbone	Memory Game	Painter	Path finding	Saved Resource	Starter Kit	Todo-Backbone	Todo-Ember
(a) Total	833	794	1,522	1,458	1,075	3,636	905	4,458
(b) Parsed	155	279	419	328	197	1,578	241	1,734
(c) Cold	146	218	322	284	190	1,380	231	1,506
(d) Warm	7	52	67	35	6	148	8	161
(e) Hot	2	9	30	9	1	50	2	67

that the parsing overhead would be substantial, as seen in Figure 1, unlike the JITC overhead, which is controlled via adaptive compilation with multi-tier architecture.

This article proposes *concurrent parsing* of JavaScript, inspired from *concurrent compilation* [Krintz et al. 2001; Ha et al. 2009; Böhm et al. 2011], which concurrently performs dynamic compilation on different threads to hide its overhead. Concurrent parsing performs the parsing of JavaScript functions in advance on different parsing threads, while the main thread is executing the parsed functions. When performed on multi-cores, this can hide the parsing overhead from the main execution thread, reducing the JavaScript execution time, thus reducing the overall app loading time.

Although the idea is simple, there are a couple of non-trivial issues involved. First, we need to separate the JavaScript parsing job from the main thread correctly and make it run on different threads in parallel, without violating the execution semantics of JavaScript. Secondly, we need an efficient multi-threaded parsing architecture, which can effectively reduce the synchronization overhead while scheduling the parsing requests in a timely manner. Finally, we need to choose the parsing targets properly among the functions included in the app, since there will be no benefit if we parse non-executing functions in advance.

We handled these issues and implemented concurrent parsing on the JavaScript engine of the WebKit browser. The contributions of this article are as follows:

—We discovered a bottleneck related to JavaScript parsing in web app loading and proposed concurrent parsing to reduce the bottleneck.
—We proposed an efficient multi-threaded parsing architecture, which handles parsing requests in parallel without violating the JavaScript semantics.
—We explored two methods of choosing parsing targets: one based on profiled information on a previous run and the other based on speculative heuristics.
—Our implementation on a commonly used web browser showed a tangible reduction of the whole loading time for various real web apps.

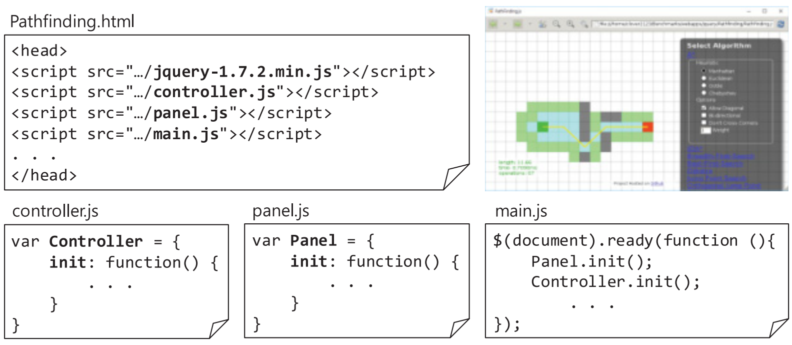
The rest of this article is structured as follows. Section 2 describes the background on web apps and modern JavaScript engines. The proposed technique of concurrent JavaScript parsing is depicted in Section 3. Section 4 shows our implementations on the WebKit browser. Our experimental results are in Section 5, and related work is in Section 6. Section 7 has the summary.

2. BACKGROUND

In this section, we briefly review how the web app works and how the JavaScript engine works. We also describe how the JavaScript parsing works.

2.1. How Web App Works

Web app is programmed using HTML, CSS, and JavaScript, as in a regular web page. Figure 2 shows an example of a web app, which searches the shortest path between two points using a chosen algorithm. The HTML file includes tags to express the web



Pathfinding.html
```
<head>
<script src="…/jquery-1.7.2.min.js"></script>
<script src="…/controller.js"></script>
<script src="…/panel.js"></script>
<script src="…/main.js"></script>
. . .
</head>
```

controller.js
```
var Controller = {
    init: function() {
        . . .
    }
}
```

panel.js
```
var Panel = {
    init: function() {
        . . .
    }
}
```

main.js
```
$(document).ready(function (){
    Panel.init();
    Controller.init();
        . . .
});
```

Fig. 2. Pathfinding web app, which searches the shortest path between points based on several algorithms.

components, some of which are *script* tags. Each script tag corresponds to a JavaScript code, often residing in a separate file. Figure 2 shows the HTML file including four script tags. JavaScript framework such as *jQuery* is often included in the first script tag as in Figure 2, so that the following script tags can use its library functions (e.g., *ready()* function used in main.js to register an event handler).

Generally, the execution of a web app is composed of two steps: *app loading* followed by *event-driven computation*. During the app loading, the browser reads the HTML file and parses all of its tags to build a DOM tree, which is then displayed on the screen based on CSS by the rendering engine. Whenever a script tag is encountered during the HTML parsing, the DOM construction is paused and the corresponding JavaScript code is executed before resuming the DOM construction. JavaScript code during the app loading mostly initializes objects including functions and registers them as event handlers. In Figure 2, *main.js* registers an event handler to be called when the loading completes, which initializes *Panel* and *Controller* objects by calling *init()* functions (*Panel* object represents points and paths while *Controller* object manages user inputs such as point marking or algorithm selection). The *load* event is automatically fired when the app loading is finished [Kacmarcik and Leithead 2015]. At this point, all of the required resources such as DOM tree, images, scripts have been loaded and the user sees the first screen of the web app on the browser.

After app loading, a web app works in an event-driven manner such that the JavaScript functions registered as event handlers are called when the corresponding events such as mouse clicks or timer events occur, often changing the DOM tree, which is then re-rendered for an updated display. Pathfinding app calculates and draws the shortest path when the user selects an algorithm using a click in Figure 2.

2.2. How JavaScript Engine Works

JavaScript code in a web app is executed by the JavaScript engine in the browser. Modern JavaScript engines have a multi-tier architecture to tradeoff the startup delay and the performance. For JavaScript code executed only a few times, an interpreter is used to start execution early. For frequently executed JavaScript code, JITC is used to generate high-performance code. Figure 3 shows a typical execution path of JavaScript code on modern JavaScript engines, such as WebKit's JSC [Apple, Inc. 2016b], Chrome's V8 [Google, Inc. 2016b], or Firefox's SpiderMonkey [Mozilla 2016c]. As JavaScript is distributed in source code format, the first step that the JavaScript engine must do is parsing, which translates the source code to an intermediate representation such as bytecode. Then, interpreter initially executes the JavaScript code by interpreting the bytecode (V8 skips the interpreter and directly goes to the baseline JITC). When the

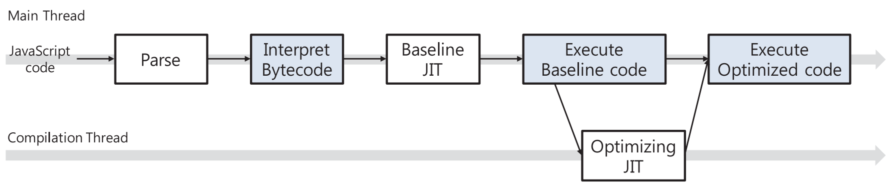


Fig. 3. Multi-tier execution architecture of JavaScript engine.

JavaScript code is found to be warm, the bytecode is compiled by the baseline JITC and finally, when it is found to be hot, the bytecode is re-compiled by the optimizing JITC.

Since the optimizing JITC performs many optimizations, its overhead is high. To reduce the overhead, modern JavaScript engines adopt *concurrent compilation* such that the optimizing JITC is performed on separate threads (compilation threads), concurrently with the main thread to hide its overhead, as shown in Figure 3. In fact, our proposed concurrent parsing works while concurrent compilation still proceeds, so there can be some competition between the parsing threads and the compilation threads, which will be discussed later.

Among the steps in Figure 3, the parsing step time takes a significant portion of the running time for the JavaScript code executed during app loading, as seen in Figure 1. This is so since a JavaScript function tends to be executed infrequently, not enough to trigger the baseline JITC or the optimizing JITC, as in Table I. Now, we will review the parsing process more in detail below.

2.3. How JavaScript Parsing Works

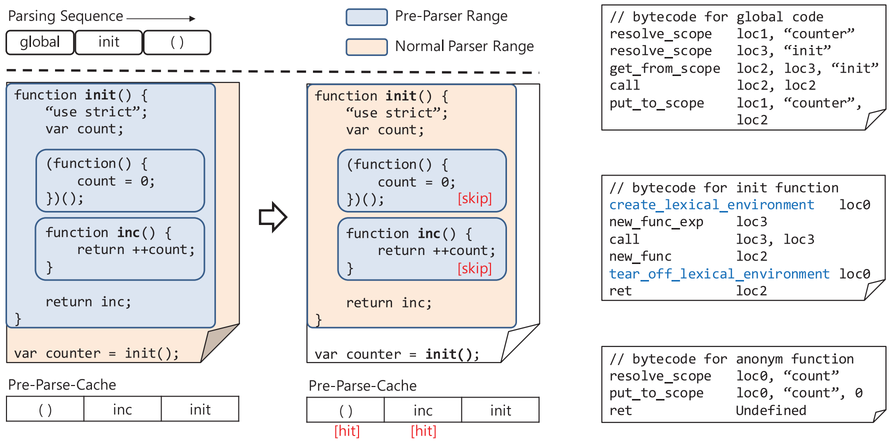
Unlike static languages such as C/C++/Java, JavaScript allows first-class functions. So a function is an object that can be passed as an argument or a return value. JavaScript also allows nested functions so that a function can be defined inside another function. Finally, JavaScript code in the web app is divided based on the script tags. JavaScript parser should deal with these diverse features.

There are three types of JavaScript code (ECMAScript6 recently added a new type called *module,* which our environment does not support, so we do not handle the module type in this article). *Global* code is the code residing in the global scope of each script tag. *Eval* code is the code supplied to the argument of the built-in *eval()* function, which is regarded as a JavaScript expression or statement, being executed dynamically at runtime (e.g., *var x = 1; eval("x + 1");* where *eval()* will return 2). Finally, *Function* code is the code in a function.

Figure 4(a) shows an example of JavaScript code in a script tag, which will be executed as follows. The Global code is executed first, which will call the *init()* function in *var counter = init()*;. The *init()* function declares a local variable *count* and initializes it to zero using the anonymous function located next (which is called at the same time it is defined). Then, the *inc()* function defined next is passed as a return value and assigned to the global variable *counter*.

The rest of Figure 4 will illustrate the parsing process, especially related to our target JavaScript engine (WebKit's JSC engine), which is explained below.

2.3.1. Parsing Process. Modern JavaScript engines adopt *lazy parsing*. That is, they perform parsing only for those functions or global code executed, just before they are executed for the first time, instead of parsing the entire code at once. There is no JavaScript rule that enforces lazy parsing (i.e., it is all right to parse all functions at once), but it is commonly used for fast startup because many JavaScript functions are not executed, as seen in Table I for app loading.



```
function init() {
    "use strict";
    var count;

    (function() {
        count = 0;
    })();

    function inc() {
        return ++count;
    }

    return inc;
}
var counter = init();
```

```
// bytecode for global code
resolve_scope    loc1, "counter"
resolve_scope    loc3, "init"
get_from_scope   loc2, loc3, "init"
call             loc2, loc2
put_to_scope     loc1, "counter",
                 loc2
```

```
// bytecode for init function
create_lexical_environment    loc0
new_func_exp     loc3
call             loc3, loc3
new_func         loc2
tear_off_lexical_environment loc0
ret              loc2
```

```
// bytecode for anonym function
resolve_scope    loc0, "count"
put_to_scope     loc0, "count", 0
ret              Undefined
```

(a) Parsing for global code (b) Parsing for *init()* function (c) Generated bytecode

Fig. 4. JavaScript parsing process. Normal parser operates on the body of called function or global code, and pre-parser operates on each encountered inner function. Normal parser performs the actual parsing, which translates the JavaScript source code to the bytecode.

JavaScript allows nested functions, which is a challenge for lazy parsing. That is, when an outer function is called and parsed, its inner functions should not be parsed yet (an inner function is parsed when it is called during the execution of the outer function), but some information on the inner functions is needed for the parsing of the outer function. To handle this issue, most JavaScript engines include two types of parsers: the *normal parser* and the *pre-parser* (they are named differently for different engines, but we use the names of V8). The normal parser performs the actual parsing, which translates the JavaScript source code to the bytecode. The pre-parser is a subroutine invoked only when the normal parser parses an outer function, in order to scan its inner functions in advance. The pre-parser just reads inner functions without any code translation and collects the information on the inner functions such as their boundaries and variables defined in them, as explained below.

For the example in Figure 4, the parsing order will be the same as the execution order. First, JSC parses the global code first by invoking the normal parser, which will generate the bytecode only for the statement *var counter = init();* in Figure 4(a), because the *init()* function is defined but not called yet. When the normal parser encounters *init()* during the parsing of global code, it invokes the pre-parser to scan this inner function. The pre-parser reads the body of *init()* to identify the function boundaries and to collect the variable information. During the pre-parsing of the *init()* function, another inner function (an anonymous function) will be encountered. At this point, the pre-parser invokes another pre-parser to handle this anonymous function. After the second pre-parser completes its handling, JSC will save the information on the anonymous function in a cache called the *pre-parse-cache* as shown in Figure 4(a). Now, pre-parsing for *init()* will resume but encounter another inner function *inc()*, thus invoking another pre-parser, which will save the information on *inc()* in the pre-parse-cache as the next item. Finally, pre-parsing of *init()* will complete, saving its information in the pre-parse-cache. Then, the control goes back to the normal parser, which will generate the bytecode for the global code as in Figure 4(c).

Then, when the *init()* function is called by executing the global code, the normal parser will parse *init()* as in Figure 4(b). At this moment, the information on *init()* is available in the pre-parse-cache, but it is of no use for the normal parser because the cached data is not enough to translate the code and normal parser should scan the function body again. Normal-parsing for the body of *init()* will encounter the anonymous function again. In this case, there already exists the cached information on the anonymous function in the pre-parse-cache, so the normal parser merely reads it and skips the invocation of the pre-parser. Similarly, pre-parsing for *inc()* function can be omitted. Finally, the normal parser will generate the bytecode for *init()* as in Figure 4(c), which will then be executed, consecutively calling the anonymous function and invoking the normal parser for it. The *inc()* function will not be parsed by the normal parser since it is not called at all.

The argument for the *eval()* function is not pre-parsed (i.e., not saved in the pre-parse-cache), but is normal-parsed when the *eval()* function is executed.

2.3.2. Parsing Semantic. Since we propose in-advance, concurrent parsing of JavaScript code, we might parse functions differently from the original parsing order. It might be questioned if this can violate any JavaScript semantics. Generally, JavaScript does not enforce any restriction on the parsing order. For example, eager parsing instead of lazy parsing does not affect the correctness. Also, there is no dependence between the code parsed earlier and the code parsed later, even for dynamically created code via *eval()*. However, there are two parsing semantics that concurrent parsing must respect, which are between outer and inner functions.

The first one is related to *strict mode*. Strict mode is for secure JavaScript programming by detecting awkward syntax as a real error, which would otherwise be accepted in non-strict mode. For example, strict mode does not allow a variable to be used without declaring it (e.g., $x = 3.14;$ without declaring x is an error). Strict mode is declared by adding *"use strict";* at the beginning of a JavaScript file or a JavaScript function. If declared at the beginning of a file, all code in the file will be executed in strict mode. If declared inside a function, only the code inside the function including its inner functions is in strict mode. In Figure 4, the *init()* function uses strict mode and accordingly its two inner functions are also in strict mode. As the strict mode is propagated from outer functions to inner functions, both the normal parser and the pre-parser need to deliver the mode information to the parsers of inner functions so that an appropriate parsing action can be made.

The other semantic is related to *closure*. In Figure 4, two inner functions defined in *init()* access the local variable *count* defined in *init()*. These special functions which access the local variable declared in its enclosing outer function are called *closure functions*, and the outer function's variable is called *closure variable*. Normally, local variables of a function on the stack are removed when the function returns, but the closure variables must be kept even after the outer function returns since they can be accessed by a closure function later (e.g., after *init()* returns *inc()* to the global scope, we can call *inc()* using *counter(),* which will access the closure variable *count*). So JavaScript engines make a *lexical environment* when executing the outer function, which records the variables created within the scope of outer function, and deliver it to the closure function. Using this lexical environment, closure functions can access the closure variables. For the parsing, the normal parser of the outer function should detect the closure variables using the pre-parser when it reads the inner functions. If an inner function uses the closure variables, the normal parser generates the bytecode to handle the lexical environment, as depicted in Figure 4(c). That is, *create_lexical_environment* generates a new lexical environment for *init()* where its local variables are recorded. Then, the inner functions receive this lexical environment. Finally, *tear_off_lexical_environment* copies its recorded data to keep it even after the outer function returns. For the inner function, which uses non-local variables such as global variables or closure variables, the normal parser simply generates a bytecode (i.e., *resolve_scope*), which accesses the non-local

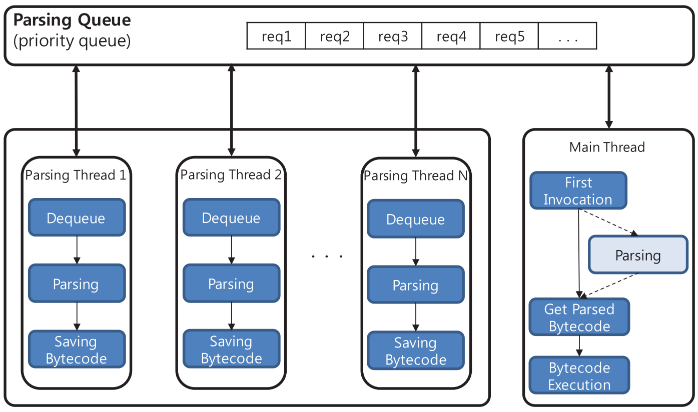


Fig. 5. Proposed concurrent parsing architecture.

variables by exploring the lexical environment. If multiple lexical environments are generated, *resolve_scope* explores them one-by-one to find a matched variable.

3. CONCURRENT PARSING APPROACH

As a single-threaded language, JavaScript global code and functions are executed in order by the main thread, and they are parsed just before being executed (we will regard global code also as function hereafter). This section describes our approach to reducing the parsing overhead, *concurrent parsing*, which performs parsing by dedicated parsing threads in advance. The main thread works as usual, except that if the function to be executed is already parsed, the main thread will execute it without parsing; otherwise, the main thread will parse it and then execute as usual. There are two main issues to discuss here. The first one is how to construct an efficient multi-threaded architecture, which coordinates the main thread and the parsing threads. Another issue is how to choose the target code for concurrent parsing for the parsing threads. This section will discuss these issues in a general context, and the next section will describe the WebKit-specific implementation.

3.1. Concurrent Parsing Architecture

Figure 5 depicts the overall architecture of our parsing system. There is a central queue structure shared between the main thread and the parsing threads, called the *parsing queue*. Parsing queue is implemented by a priority queue, where the priority depends on the way of choosing the parsing targets (based on the previous execution order, location, and scope, as will be described later). Both the main thread and the parsing threads can enqueue a parsing request for a function to the parsing queue. A parsing thread can dequeue a parsing request from the parsing queue, perform normal parsing for the request, and store the parsed bytecode ready for execution by the main thread. Since the parsing queue works as the main interface between the main thread and the parsing threads, we can add more parsing threads without changing the parsing architecture, thus gaining some scalability. When the main thread is about to execute a function for the first time, it will check the parsing queue for the parsing state of the function.

The parsing state of a function and their transition are shown in Figure 6. As soon as a parsing thread completes the parsing of a function, it changes the parsing state so that the main thread can identify it immediately. Depending on the state of a parsing request, the main thread works as follows when it is about to execute a first-time called function:

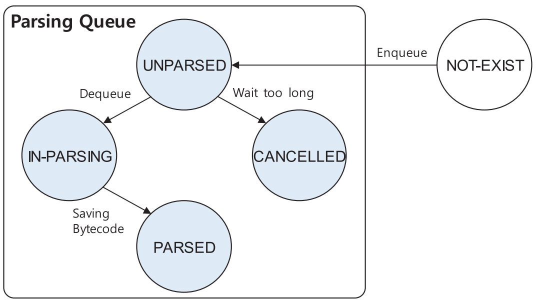


Fig. 6. State transition of a parsing request.

—NOT-EXIST – A parsing request for the function has not been enqueued yet. The main thread itself will parse the function.
—UNPARSED – A parsing request has been enqueued, but no parsing thread has dequeued and started parsing for the function yet. In this case, the main thread will immediately cancel it (convert the state to the *cancelled* state) and the main thread itself will parse the function since this will be faster than waiting until a parsing thread parses the function.
—IN-PARSING – A parsing thread has dequeued and started parsing for the function, but the parsing is not complete yet. In this case, the main thread will wait until the parsing completes.
—PARSED – A parsing thread has completed the parsing of the function. The main thread will read the bytecode and execute it directly.

This use of parsing state shared between the main thread and the parsing threads allows synchronization among them with a small overhead. Our parsing state is somewhat similar to the compiled state variables used previously for concurrent compilation [Ha et al. 2009; Böhm et al. 2011].

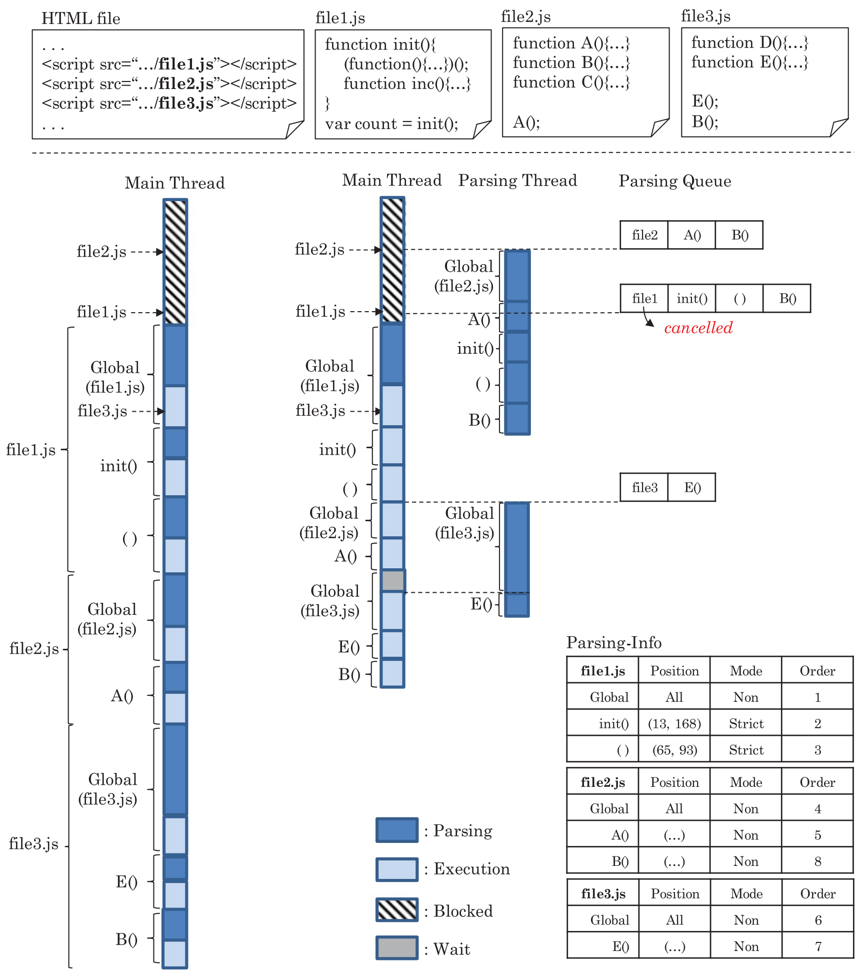
3.2. Choosing Parsing Target Functions

Now, we discuss how to choose target functions for concurrent parsing. It should be noted that as long as we respect the parsing semantics discussed in Section 2.3.2, the parsing order imposed by concurrent parsing does not affect the correctness of app execution even when it is different from the original parsing order. However, the parsing order affects the performance such that if the parsing thread can parse a function before it is executed, its original parsing overhead can be removed. Since JavaScript code in an app includes 3∼4 times more functions than those executed, concurrent parsing of arbitrary function in an arbitrary order would not parse the functions in a timely way, failing to reduce the parsing overhead. We have some guidelines for efficient selection and scheduling for target functions as follows:

—Exploit each parsing thread as fully as possible.
—Enqueue a parsing request as soon as possible when it gets available.
—Dequeue and parse a parsing request with the highest priority

We will explore two approaches, completely different, but following these guidelines.

3.2.1. Profile-Based Concurrent Parsing (PCP). The first approach is based on the profiled information from the previous run of the app. When an app runs for the first time, the browser will record the information on all the functions executed during app loading and their first-time execution order in a file, which we call *parsing-info* (so if parsing-info is missing, the browser knows that the web app is executed for the first time). We



HTML file
```
. . .
<script src="…/file1.js"></script>
<script src="…/file2.js"></script>
<script src="…/file3.js"></script>
. . .
```

file1.js
```
function init(){
    (function(){…})();
    function inc(){…}
}
var count = init();
```

file2.js
```
function A(){…}
function B(){…}
function C(){…}

A();
```

file3.js
```
function D(){…}
function E(){…}

E();
B();
```

Parsing-Info

file1.js	Position	Mode	Order
Global	All	Non	1
init()	(13, 168)	Strict	2
( )	(65, 93)	Strict	3

file2.js	Position	Mode	Order
Global	All	Non	4
A()	(…)	Non	5
B()	(…)	Non	8

file3.js	Position	Mode	Order
Global	All	Non	6
E()	(…)	Non	7

(a) Sequential parsing (b) Profile-based concurrent parsing (PCP)

Fig. 7. Example of sequential parsing and profile-based concurrent parsing (PCP).

will use parsing-info for the next runs of the app so that the parsing queue includes the parsing requests only for those functions in parsing-info, with the priority equal to the recorded order. Even if the next runs execute different functions or in a different order from the parsing-info, the correctness is not violated, as we mentioned previously. Fortunately, the same functions are likely to be executed in the same order during app loading, since the app loading tends to repeat the same job, such as the framework initialization or app initialization. So, this approach, which we call PCP, is somewhat ideal case and would show the best performance.

Figure 7 shows how PCP works for the example JavaScript tags, and compares the run of the original sequential parsing and the PCP. There are one HTML file and three script files where *file1* is a simplified form of Figure 4(a). Sequential parsing, the way of current JavaScript engines running, parses and interprets the function on the main thread in the order of first-time execution as shown in Figure 7(a). One thing to note is that before executing a JavaScript tag, the browser should fetch its JavaScript file from the web server or the local disk. To reduce the file fetching overhead, modern browsers employ *pre-loader* [Koivisto 2008; Cascaval et al. 2013]. While the browser is blocked for fetching the first script file, pre-loader scans the rest of the HTML tags looking for other resources including the script files that need to be fetched. The pre-loader then starts downloading these resources using a separate process in the browser. In Figure 7(a), fetching for *file2* and *file3* starts by the pre-loader while *file1* is being fetched, so their fetching overhead can be hid. In fact, depending on the size and the network traffic, the order of fetch completion can be different from the order of fetch start, so fetching of *file2* is assumed to be completed earlier than fetching of *file1* in Figure 7(a).

Figure 7(b) shows an example of PCP running with one parsing thread. It also shows the parsing-info for the JavaScript code, which contains the location, the mode (i.e., strict or non-strict), and the execution order for each function executed during app loading. Using the location stored in parsing-info, a parsing thread can identify the parsing range, and using the mode, it can take an appropriate parsing action. Finally, using the execution order, each request has a priority value and sorted in the parsing queue. We do not record the information on closure variables in the parsing-info, since detecting the closure variables will be done by the pre-parser as usual, when the parsing thread do normal parsing for the outer functions. In Figure 7(b), parsing-info for *file1* have three elements, the global code, the *init()* function, and the anonymous function, which were executed during app loading. The main thread will read the parsing-info to fill the parsing queue at the beginning of app loading.

When a script file is fetched, the browser reads the parsing-info of this file (actually, parsing-info is organized as a unit of the script file) and enqueues the parsing requests for its elements immediately. In Figure 7(b), *file2* is fetched first by the pre-loader, then the parsing requests for its global code, *A()*, and *B()* are enqueued and sorted based on the execution order. Then, the parsing thread dequeues and parses a request one-by-one (so, global code and *A()* are parsed). When *file1* is fetched, parsing requests for its global code, *init()*, and the anonymous function are also enqueued. The parsing queue rearranges its elements based on the execution order, so it now has global code of *file1*, *init()*, the anonymous function, and *B()* in this order as shown in Figure 7(b). This time, however, before global code of *file1* is dequeued by the parsing thread, it is assumed to be executed by the main thread. Its state is unparsed, thus being cancelled, and the main thread itself parses the global code. The remaining functions of *file1* (*init()* and anonymous) will still be parsed by the parsing thread in time, before they are executed by the main thread.

The fetching of *file3* is done during the execution of *file1*. So the browser waits until the JavaScript engine finishes executing *file1* and returns control back to the browser. Then, the browser enqueues the requests for *file3*, which will be parsed by the parsing thread as usual. Now, when the *file3* is to be executed, the request of its global code is *in-parsing* state, so the main thread should wait until the parsing completes. This will cause the *wait time* depicted in Figure 7(b).

The wait time or the parsing cancellation often occurs for the framework script files since they are huge compared to other files, which will be discussed in the experimental results. Despite the wait time or cancellation, Figure 7(b) indicates that PCP can remove much of the original parsing overhead of Figure 7(a), reducing the JavaScript execution time tangibly, if the parsing behavior of the previous run (i.e., parsing-info) repeats in the current run.

3.2.2. Speculation-Based Concurrent Parsing (SCP). The other approach, speculation-based concurrent parsing (SCP), does not resort to any profiled information but dynamically chooses the parsing targets. There are three simple heuristics that we use.

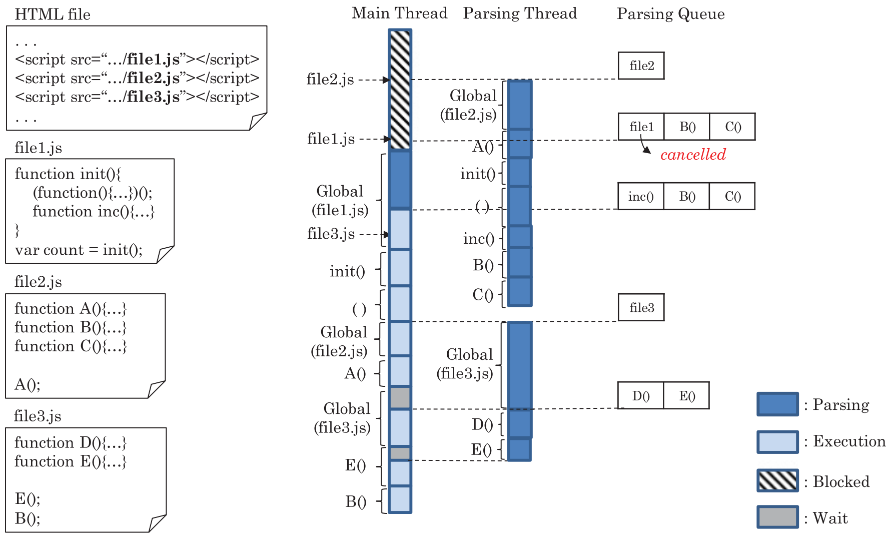


Fig. 8. Example of speculation-based concurrent parsing (SCP).

One is that those functions located in the earlier script tags should have higher priority than those functions located in later script tags. Another one is that outer functions should have higher priority than inner functions since inner functions will be executed only after outer functions are executed. Finally, we choose large functions only (whose size is bigger than a threshold value) since we found that large functions are more likely to be executed during app loading.

We describe how SCP enqueues the parsing requests. The parsing request for the global code (script file) is enqueued to the parsing queue when each file is fetched, exactly as in PCP. When the global code of a script is dequeued and parsed, *all* large-sized functions read by the pre-parser will be added to the parsing queue, unlike PCP. All parsing requests in the parsing queue are prioritized based on the file order of their script files and the function depth in the outer-inner function hierarchy in each file. To respect the parsing semantics discussed in Section 2.3.2, SCP passes the mode information detected from outer functions when it enqueues each inner function. Closure semantic will be handled by the pre-parser as in PCP.

Figure 8 shows an example of SCP running with one parsing thread, using the same execution scenario of Figure 7 (in this example, we assume that every function is large enough to be a target of concurrent parsing). When *file2* is fetched first, the parsing request for its global code is enqueued. When the normal parser parses the global code in the parsing thread, the pre-parser will read the inner functions and enqueue each function immediately. After the global code of *file2* is parsed, the parsing queue has three elements, *A()*, *B()*, and *C()*.

When *file1* is fetched, the request of its global code is enqueued. As in PCP of Figure 7, when *file1* is about to execute, the parsing thread is busy in parsing *A()* so the main thread cancels the request and parses it. During the parsing of *file1*, the anonymous function, *inc*(), and *init*() are newly enqueued by the pre-parser in this order, and the parsing queue is sorted based on the file order and then the depth. The parsing requests will be dequeued to the parsing thread in this sorted order.

When *file3* is enqueued by the browser, the main thread should wait for the parsing of its global code before executing *file3*, exactly as in PCP. Also, the main thread should wait for the parsing of *E()* since it is *in-parsing* state when the main thread is about to

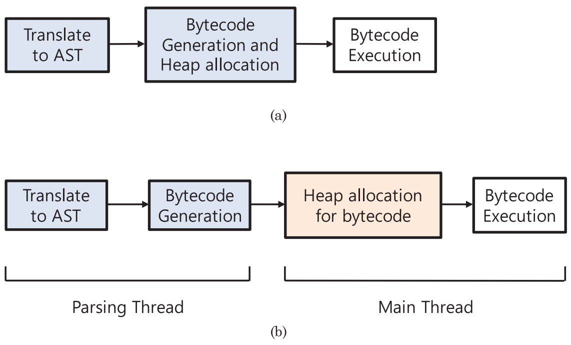


Fig. 9. Separation of parsing process in JSC.

execute it. This is caused by *D(),* parsed earlier than *E()*, although *D()* is not executed at all. Since SCP might parse a function not to be executed unlike PCP, SCP can suffer from more wait time than PCP. Also, due to its size-based heuristic, SCP might not parse a small function to be executed, leaving its parsing overhead to the main thread. So, SCP would work worse than PCP in hiding the parsing overhead, as Figure 8 shows compared to Figure 7(b).

4. CONCURRENT PARSING ON WEBKIT

We implemented our proposed concurrent parsing on top of the WebKit browser [Apple, Inc. 2016a] and the JavaScriptCore (JSC) engine [Apple, Inc. 2016b], a commonly employed platform in Apple's Safari browser [Apple, Inc. 2016c] or Tizen [Tizen 2016]. We discuss some of the implementation issues here.

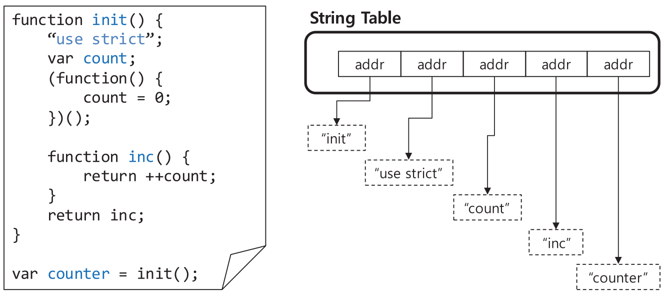
4.1. Separation of Parsing Process

In JSC, a function is parsed first before being executed unless it has already been parsed. The parsing job is divided into two phases: translation to *Abstract Syntax Tree* (AST) and bytecode generation. AST is an intermediate tree structure generated during parsing. Bytecode for each function is generated from the function's AST. Then, the JSC engine executes the bytecode via interpretation initially and via JITC as it gets hot. This parsing/execution procedure is depicted in Figure 9(a).

In order to separate the parsing job from the main thread to the parsing thread, we need to handle the memory allocation issue for AST and bytecode. The JSC engine has a single *JavaScript heap* where all JavaScript objects and their related helper objects are allocated. To avoid the complexities in implementing concurrent memory allocations, we make the heap allocation be done only by the main thread.

One memory issue related to parsing is that the bytecode of each function is an object needed to be allocated to the JavaScript heap, while the AST is not. So, we divide the bytecode generation in Figure 9(a) into two phases, bytecode generation and heap allocation for bytecode, where the former is done by the parsing thread while the latter is done by the main thread, as depicted in Figure 9(b). So, a parsing thread handles translation to AST and bytecode generation, without any JavaScript heap allocation. And, the main thread gets the parsed bytecode and allocates it to the heap before executing it. In fact, there are other constant JavaScript objects such as number or string objects that the parser needs to allocate together with the bytecode, so a parsing thread records the information on all these objects, and the main thread uses it for heap allocation later.

There is one difference for memory allocation between AST and bytecode. Once an AST is created and used to generate the bytecode by a parsing thread, it is no longer



```
function init() {
    "use strict";
    var count;
    (function() {
        count = 0;
    })();

    function inc() {
        return ++count;
    }
    return inc;
}

var counter = init();
```

Fig. 10. String table managed in JSC engine.

needed. On the other hand, the bytecode needs to be kept until the main thread takes it away. So we allocate a dedicated space for AST for each parsing thread (and for the main thread as well) and reuse it for each function parsed in the thread. This technique, known as *AST pool allocation* [Lu et al. 2006; Barenghi et al. 2012], has been used for parsing of divided code chunks in parallel, and we employ it for concurrent parsing in JavaScript. This technique is known to increase the data locality as well as to prevent the parsing threads from being serialized during the memory allocation of AST, since repeated normal allocation using *malloc()* for AST will incur substantial synchronization overhead between the main thread and the parsing threads.

4.2. Shared Data Structure

We need to share two data structures between the main thread and the parsing threads, in addition to the parsing queue. The first one is the string table and the other is the pre-parse-cache. The string table is needed for correct execution while the pre-parse-cache is needed for parsing performance.

JSC maintains its own string table for identifiers and constant strings as in Figure 10. Each character string is wrapped into a unique *StringImpl* object and the address of StringImpl is saved in the string table. JSC represents and recognizes each string by StringImpl object internally. Using the string table, JSC could prevent duplications of the same string value. When a parser encounters a new identifier during parsing process, it generates a StringImpl object for the new string and adds the address value in the string table. By default, the string table is accessible only by the main thread. To make this string table accessible by the parsing threads, we share the string table using *mutex*, so that adding a new string value to the string table is serialized to make a unique StringImpl object for each string value.

As described in Figure 4, the pre-parser stores its pre-parsing result in the pre-parse-cache for later reuse. We made the pre-parse-cache be shared between the main thread and the parsing threads by serializing their accesses. When a normal parser in a parsing thread needs to invoke the pre-parser, it locks and looks up the pre-parse-cache for the matching inner function. Similarly, when a pre-parser finishes pre-parsing, it locks and stores the pre-parsing result in the pre-parse-cache.

4.3. Source Code Hashing

Web app is generally delivered as source code format from the web server. If the source code of web app is modified by app developer, browser on client-side could not recognize the change and our PCP approach would be useless, potentially incurring an incorrect execution. We can distinguish each JavaScript code based on its URL address but it is not enough to detect the modification of source code.

Table II. Description of Web Apps

Web App	Description	Framework
Hello-Backbone	Simple tutorial of "hello world" example (http://arturadib.com/hello-backbonejs/1.html)	Backbone.js, jQuery, Underscore.js
Memory-Game	Memory game of finding a matching pair of cards (http://igorminar.github.io/Memory-Game/app/index.html)	AngularJS
Painter	Web painter (http://lislis.sakura.ne.jp/canvas/paint/paint.html)	jQuery
Pathfinding	Find the shortest path between two points (http://qiao.github.io/PathFinding.js/visual/)	jQuery
Saved Resource	Add or remove an item (http://fiddle.jshell.net/dLwkqbmt/show/light/)	CanJS, jQuery
StarterKit	Starter kit for emberJS framework (https://github.com/emberjs/starter-kit)	Ember.js, jQuery
Todo-Backbone	Todo list app with backboneJS framework (http://todomvc.com/examples/backbone/)	Backbone.js, jQuery, Underscore.js
Todo-Ember	Todo list app with emberJS framework (http://todomvc.com/examples/emberjs/)	Ember.js, jQuery, Handlebars.js

To resolve this issue in PCP, we adopt hashing scheme for each JavaScript code. We cache the calculated hash value of each JavaScript code together with parsing-info. When our system tries to concurrently parse each global code, it first calculates the hash value of the current source code and compares it with the cached value. If the two hash values are identical, there is no modification and we parse that code concurrently. Otherwise, if modification has been occurred, we discard the cached data of the code and run the parsing process in main thread as usual. After the loading completes, we newly record the parsing-info of the modified code.

5. EVALUATION

5.1. Experimental Setup

We ran our experiment on an ODROID-C1+ ARM board [Hardkernel Co. 2016], equipped with Cortex-A5 1.5Ghz quad-core CPU and 1GB RAM. There is a 16GB SD card for use as the local storage. The ARM board is running Ubuntu 14.04 and a WebKit (revision 174059). We experimented with 8 web apps from various domains, all of which are programmed with one or more JavaScript frameworks, as listed in Table II.

We evaluated three types of parsing configurations: original sequential parsing, profile-based concurrent parsing (PCP), and speculation-based concurrent parsing (SCP). There is a single main thread for the original parsing. For PCP and SCP, we experiment with one, two, three, four, and five parsing threads, along with one main thread, to check the scalability issue. There are only four cores in the CPU, so experimenting with more than four threads (three parsing threads + main thread) is meaningless, especially since the browser includes additional threads. Also, the WebKit's JSC engine runs with concurrent JITC threads, so competition among threads will be even higher. In fact, we found that we can achieve the best performance with *two* parsing threads, which will be presented and analyzed shortly.

We considered the cases of *install-based platform* for web apps where all source files including frameworks reside locally, such as Tizen (mobile jQuery) or webOS (Enyo). So, we downloaded each web app and tested it offline on the board by fetching the app from local storage. In fact, this offline-based evaluation measures more consistent impact of concurrent parsing without suffering from network fluctuation.

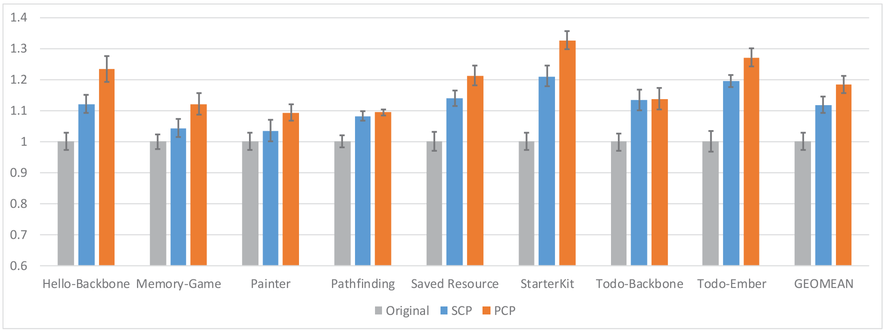


Fig. 11. Speedup of app loading by SCP and PCP with two parsing threads.

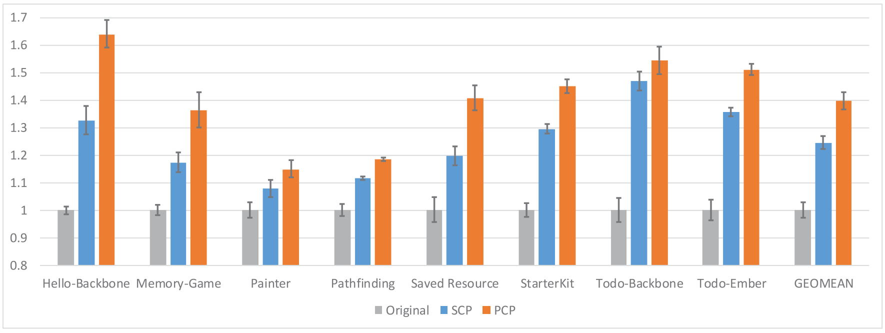


Fig. 12. Speedup of JavaScript execution by SCP and PCP with two parsing threads.

We measured the entire app loading time from the start of web app and until the *load* event fires. We measured the loading time 10 times and took an average, with the error bound being calculated. For PCP, we obtained the parsing-info of each app by running it once in advance. For SCP, we defined the size threshold as 300 bytes, so only those functions larger than 300 bytes are enqueued to the parsing queue.

5.2. Performance Analysis

Figure 11 shows the performance improvement of the app loading time obtained by SCP and PCP with two parsing threads, compared to the performance of original parsing as a basis of 1.0 (with error bounds depicted on each bar). The average performance improvement is 11.8% (SCP) and 18.2% (PCP). As expected, PCP achieves much better performance than SCP since it knows which functions will be executed and parsed. The performance impact is the highest for the *StarterKit* app, which suffers the biggest parsing overhead, as seen in Figure 1. Other apps also show a similar performance gain proportional to their parsing overhead.

Figure 12 shows the performance improvement for the JavaScript portion of the app loading time. Both SCP and PCP improve the JavaScript performance by 24.6% and by 39.7%, respectively. This result indicates that JavaScript execution time takes a substantial portion of app loading time, and that parsing takes a significant portion of JavaScript execution. Consequently, accelerating JavaScript parsing using the parsing threads as proposed in this article is well justified.

To analyze the performance impact more specifically, we break down the app loading time of the main thread as in Figure 13. For the original parsing bars, the bottom part shows the parsing overhead of the main thread. For SCP and PCP bars, the bottom

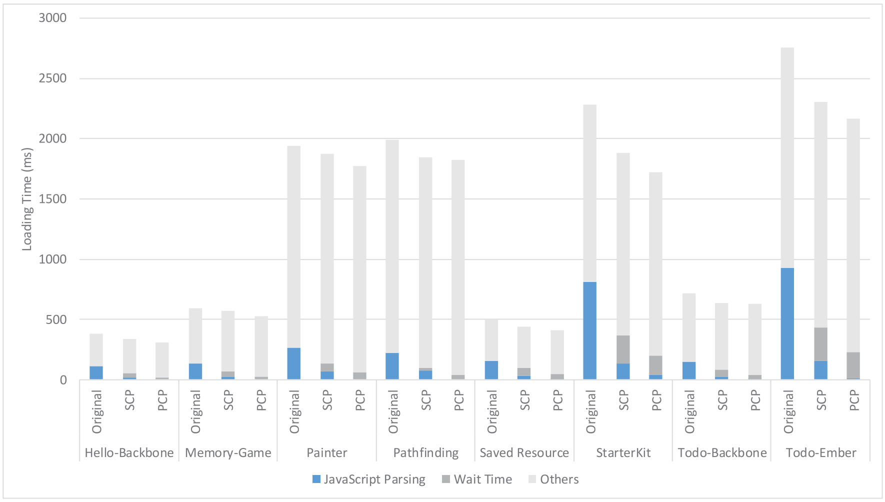


Fig. 13. Breakdown of app loading time for the main thread.

part is divided into the parsing overhead of the main thread and the wait time of the main thread, where the latter is the time for the main thread to wait since the function to be executed is *in-parsing* state by a different parsing thread.

Figure 13 indicates that concurrent parsing removes much of the original parsing overhead. The main thread rarely performs parsing, but it sometimes needs to wait to execute a function until a parsing thread parses the function. We found that most of this wait time is due to the parsing overhead of the JavaScript framework. Since the size of framework is huge compared to other JavaScript files, its parsing time is substantial. Also, the framework script tags are often located at the upper script tags in the HTML file, so they should usually be executed right after they are fetched, unlike other script tags that have some leeway between when they are fetched and when they are executed; parsing can be done during that time on a parsing thread, so the main thread does not have to wait. In Figure 13, *StarterKit* and *Todo-Ember* suffer especially from the wait time. Both apps include one huge framework file (ember-1.10.0.debug.js and ember.js), even larger than the total sum of other files. So the main thread should wait a long time for parsing the code of the framework. On average, the wait time for frameworks takes 67.2% of the entire wait time in SCP, and 76.2% in PCP.

Actually, PCP consistently have smaller wait time and parsing time than SCP in Figure 13, and these shorter times are the key factor for better performance of PCP than SCP.

5.3. Scalability

We measured the average performance by increasing the number of parsing threads from one to five, as shown in Figure 14. We found that having two parsing threads achieves the best performance for both SCP and PCP, and the performance goes down as we move from three to five threads. This is partially due to the overhead of thread creation and synchronization, but we found that it is mainly due to the contention between the parsing threads and the additional process/thread of the browser. The WebKit browser includes at least two processes for responsive and robust browsing [Apple, Inc. 2011].

The first one is the base UI process, which runs UI and manages the web processes. It also handles the resource fetching mentioned in Section 3. The second one is the

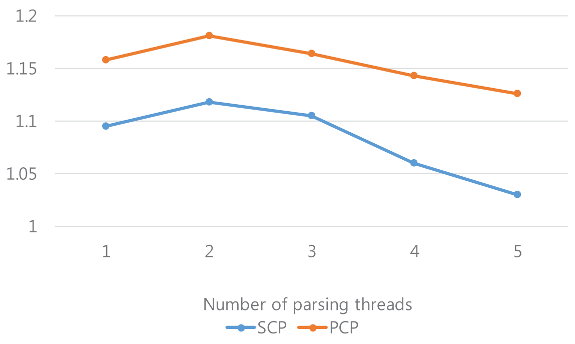


Fig. 14. Average speedup with a different number of parsing threads.

Table III. Total Number of Parsing Requests Handled in the Main Thread

	Hello-Backbone	Memory Game	Painter	Path finding	Saved Resource	Starter Kit	Todo-Backbone	Todo-Ember
Origin	155	279	419	328	197	1578	241	1734
SCP-1	113.3	175.5	280.1	234.4	137	849.7	147.5	938.3
SCP-2	91.9	160	268.3	212.5	120.9	746.3	128.4	828.1
SCP-3	90	160	267.2	209	117.5	724	124.3	808.2
SCP-4	90	160	266.1	209	117	724	124	808
SCP-5	90	160	266	209	117	724	124	808
PCP-1	0.0	0.0	1.7	0.0	1.0	2.0	0.2	1.5
PCP-2	0.0	0.0	0.0	0.0	1.0	2.0	0.1	1.2
PCP-3	0.0	0.0	0.0	0.0	1.0	1.0	0.1	0.8
PCP-4	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.1
PCP-5	0.0	0.0	0.0	0.0	1.0	1.0	0.1	1

Table IV. Size of the Parsing-info Compared to the Total Size of Web App

	Hello-Backbone	Memory Game	Painter	Path finding	Saved Resource	Starter Kit	Todo-Backbone	Todo-Ember
ParsingInfo (KB)	4.4	9.7	11.9	8.4	5.5	54.4	6.6	60.9
Web App (KB)	303.9	449.4	375.9	787.4	392.6	2,155.7	371.5	2,603.9
Overhead	1.5%	2.2%	3.2%	1.1%	1.4%	2.5%	1.7%	2.4%

web process, which handles HTML parsing and running JSC. Also, the JSC engine uses additional threads for GC and concurrent compilation. So, the contention between the parsing threads and existing processes/threads would get higher as we add more parsing threads, possibly slowing down the main thread.

Table III shows the total parsing requests handled by the main thread in each app, for each number of parsing threads in Figure 14 (average of 10 runs). Original parsing executes every parsing request in the main thread. Among these requests, PCP can handle almost all of them in the parsing threads. SCP can handle approximately less than the half of total parsing requests in the parsing threads, and this does not improve when using more than two parsing threads, as seen in Table III.

5.4. Overhead of PCP

PCP includes two extra overheads, the space overhead for parsing-info storage and the time overhead for hashing. PCP stores the parsing-info in local storage such as disk or flash memory. Table IV shows the size of parsing-info compared to the size of web app.

Table V. Hashing Overhead Compared to the Entire Loading Time of Web App

	Hello-Backbone	Memory Game	Painter	Path finding	Saved Resource	Starter Kit	Todo-Backbone	Todo-Ember
Hashing (ms)	1.3	2.1	1.5	1.3	1.6	9.7	1.6	11.9
Overhead	0.4%	0.4%	0.1%	0.1%	0.4%	0.6%	0.3%	0.5%

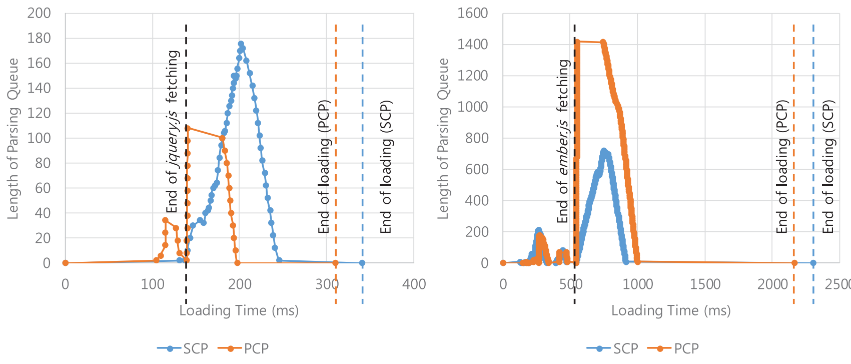


Fig. 15. Length of the parsing queue over the app loading time in SCP and PCP.

We found that parsing-info size takes almost less than 3% of the web app size, so it is not a big overhead.

The hashing overhead is involved with scanning the entire source code to calculate the hash value, yet hashing itself is a simple task composed of several bit operations. We found that the hashing overhead affects the loading time negligibly (less than 1%), as in Table V.

5.5. Length of the Parsing Queue

We examined how the length of the parsing queue changes over the app loading time. We chose *Hello-Backbone* and *Todo-Ember*, which has the shortest and longest loading time, respectively. Figure 15 shows the length of the parsing queue in SCP and PCP with two parsing threads. After the fetching of each script file, the parsing queue grows gradually in SCP as the functions are enqueued during the parsing of the global code. On the other hand, the parsing queue grows rapidly in PCP because functions are enqueued at once by reading the parsing-info right after the file fetching. Both apps show the peak length after fetching of the JavaScript framework files (*jquery.js* and *ember.js*).

The average size of a function parsed by concurrent parsing is 2.9KB (SCP) and 2.7KB (PCP). SCP shows a larger parsing unit size due to its size-based heuristic. This average size includes the frameworks, which usually include a single, huge anonymous function. Other than this anonymous function, a parsed function's size is often less than 1.0KB.

5.6. Performance Impact on JavaScript Benchmark

In addition to the previous experiments on web apps, we evaluated concurrent parsing with the JavaScript benchmark. Figure 16 shows the performance of SCP and PCP on Octane benchmark suite [Google, Inc. 2016c] with two parsing threads, which is 4.4% and 7.5%, on average, respectively. For SCP, we choose every functions for concurrent parsing, instead of only large functions, to get a maximum performance because the benchmark behaves differently than web apps. There is tangible speedup for a couple

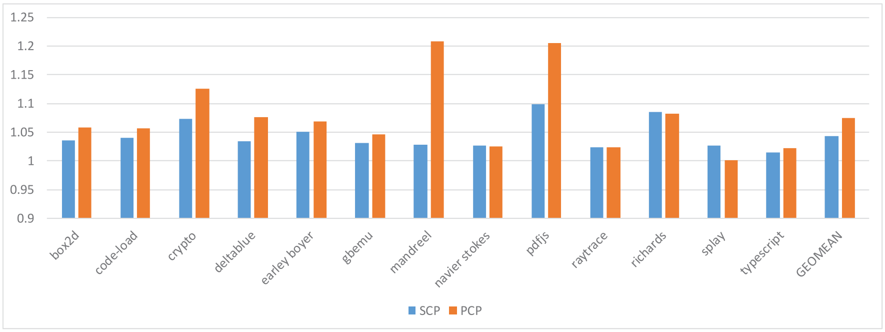


Fig. 16. Speedup of Octane benchmark by SCP and PCP with two parsing threads.

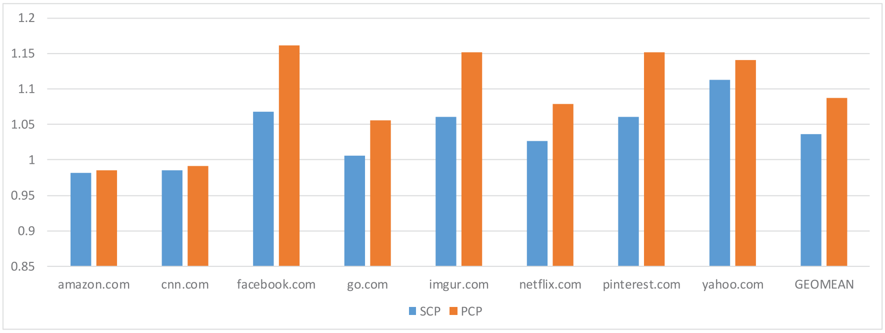


Fig. 17. Speedup of web page loading by SCP and PCP with two parsing threads.

of programs but little speedup for the others, so the overall performance impact is lower than in web apps.

The reason is that the parsing overhead in JavaScript benchmarks is smaller than in web apps, since benchmark loops and functions tend to be executed more repeatedly than web apps [Ratanaworabhan et al. 2010; Richards et al. 2010], spending more time for execution. In fact, PCP is slower than SCP in *splay* since PCP suffers from the I/O overhead of reading its parsing-info, which is higher than the benefit of PCP if the parsing overhead itself is too small.

5.7. Performance Impact on Web Pages

We also evaluated concurrent parsing for 8 web pages from the Alexa list [Alexa 2016], which have a relatively large JavaScript portion. Unlike the previous app experiment where all source files are located locally at the client, we perform this experiment online, so we load each web page from the server and run with concurrent parsing. Figure 17 shows the speedup of web page loading by SCP and PCP with two parsing threads, which is 3.7% and 8.7%, on average, respectively. Network delay and thread contention for resource loading reduce the performance impact of concurrent parsing, yet it is still tangible. For two web pages (*amazon* and *cnn*), concurrent parsing even slows down because both pages are composed of many resources including image files, so severe contention between the UI process and parsing threads seems to affect the loading time negatively. This result complements previous scalability evaluation in Section 5.3 and indicates that concurrent parsing would benefit only when there are sufficient cores to run the parsing threads concurrently.

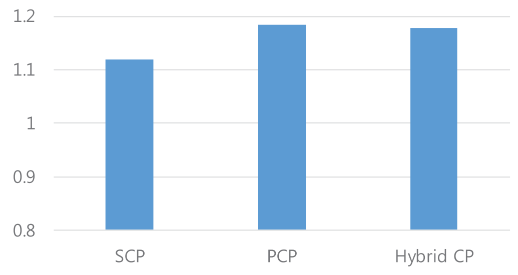


Fig. 18. Average speedup of app loading by hybrid CP, compared to SCP and PCP.

We also repeated the app experiment, yet with the frameworks loaded from the servers. We obtained a slightly lower performance compared to the offline experiment.

5.8. Hybrid Approach to Concurrent Parsing

PCP requires a profile run, yet its parsing target selection based on the previous parsing information is likely to be accurate. SCP does not require any profiled information, but its selection based on heuristics cannot be precise. To make a compromise between the two extreme approaches, we can think of a *hybrid* approach. Generally, JavaScript frameworks provide many library functions, yet only a fraction of them are used more often than others. For example, *jQuery* provides many library functions, but those functions called for searching, inserting, or deleting DOM elements using $*()* or registering event handlers using *ready()* are used most often. To exploit this behavior, we can provide some usage information for each JavaScript framework so that only those heavily used functions are parsed by PCP when we parse the framework tag, while other script tags are handled by SCP.

We emulated this proposal for the PCP run, by performing PCP only for the JavaScript framework tags, while other tags are parsed sequentially as in the original run. Figure 18 shows the result with two parsing threads, compared to SCP and PCP results. The emulated, hybrid CP is shown to achieve the most of the performance of PCP, which looks promising. However, the real usage information for a framework tag would be less precise than PCP's parsing-info, which we used here for emulation, because the usage should be generally applicable to any apps that uses the framework. It remains as future work to choose an appropriate set of functions for each framework and decide the order to parse them.

6. RELATED WORK

6.1. Web Loading Acceleration

Recently, some researches have been done to improve the web loading performance.

Chiu [2009] described a method to reduce startup latency by commenting out those JavaScript blocks not needed in app loading in advance, so that they are not parsed during app loading. To execute those blocks after loading, the app strips out those commented blocks and calls them by *eval()*. We believe concurrent parsing can complement this approach by parsing the uncommented JavaScript code in advance.

HPar [Zhao et al. 2013] proposed parallelizing HTML parsing to minimize the browser's response time. They focused on the HTML parsing and explored two kinds of parallel HTML parser: *pipelining parallelism* and *data-level parallelism*. From the evaluation on real web pages, data-level parallelism, which divides the HTML tags into chunks and parses in parallel, showed tangible performance gain.

Chrome browser recently announced a new technique, script streaming for fast page startup [Osmani 2015]. Script streaming parses JavaScript on a separate thread as soon as the download begins, allowing parsing to complete soon after the download has completed. Script streaming is similar to our work in that both parse JavaScript in separate thread for faster loading. But script streaming is only applied to *async* and *defer* scripts based on the expectation that these scripts would have large code size. *async* or *defer* scripts are used to reduce the latency incurred by blocking JavaScript in loading time. Moreover, *async* and *defer* scripts are rarely used due to their restrictions such as out-of-order execution or excluding the access to DOM tree. For example, web apps used in our evaluation contain only one *async* script among entire apps. On the other hand, our approach can be applied to every JavaScript code and concurrently parses global and function code both while script streaming only parses each global code in advance.

Oh and Moon [2015] proposed a technique to accelerate app loading based on snapshot. It caches the initialized JavaScript heap objects generated during app loading in a form of snapshot, and restores the heap from the snapshot to skip the entire app loading including the JavaScript parsing. Although snapshot has a much better performance impact, but it suffers from the space overhead (which is around 4 times of the app size), excessively larger compared to our parsing-info in PCP.

Ahead-of-time compilation (AOTC) [Park et al. 2015] saves the bytecode or the machine code in advance to remove the parsing or compilation overhead. The space overhead of AOTC for bytecode or machine code is still much larger than concurrent parsing (2.3 times and 15.4 times of the app size, respectively). So, when the storage on the embedded device is scarce as in smart TVs or smart phones, our concurrent parsing would be more suitable.

6.2. Concurrent Compilation

A number of studies have been explored that decouple the JITC from the main thread and execute it in separate threads.

Ha et al. [2009] presents a concurrent trace-based JITC for JavaScript. This article uses a *compiled state variable* to indicate the translation state of a trace, which is somewhat similar to our parsing state.

Krintz et al. [2001] proposes profile-driven, background compilation. Profiled information related to the execution time of each functions is used to prioritize functions as candidates for concurrent compilation. This approach is similar to our PCP method, which also uses profiled information of called functions.

The other approach [Böhm et al. 2011] presents trace JITC with parallel task farm composed of several compilation threads. They fully exploit the available parallelism by compiling several hot traces in a concurrent task farm, which is analogous to our concurrent parsing architecture. This work also adaptively adjusts the threshold of trace selection based on the number of waiting tasks to avoid the event of overloaded tasks, which can be used for our SCP heuristics in the future.

7. SUMMARY

To the best of our knowledge, this article is the first research work for concurrent parsing of JavaScript to improve the app loading performance. We proposed an efficient multi-threaded parsing architecture, with two methods to select the parsing targets. By decouplinjhg the parsing job from the main thread and scheduling them efficiently, we could improve the whole app loading performance by 18.2%, on average, and by 32.7% on maximum for real web apps.

Future web apps will be substantial with heavy JavaScript frameworks (e.g., Enyo in webOS [LG, Inc. 2016]), so their parsing overhead would be critical. We expect that the proposed concurrent parsing can make more tangible difference for these apps.

REFERENCES

Alexa. 2016. The Top 500 Sites on the Web. Retrieved July 10, 2016, from https://www.alexa.com/topsites.

Apple, Inc. 2011. WebKit2 - High Level Document. Retrieved July 10, 2016, from https://trac.webkit.org/wiki/WebKit2.

Apple, Inc. 2016a. WebKit. Retrieved July 10, 2016, from https://webkit.org/.

Apple, Inc. 2016b. WebKit JavaScriptCore. Retrieved July 10, 2016, from https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore.

Apple, Inc. 2016c. Safari browser. Retrieved July 10, 2016, from http://www.apple.com/safari.

Alessandro Barenghi, Ermes Viviani, Sefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella. 2012. Papageno: A parallel parser generator for operating precedence grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE'12)*. Springer, Berlin, 264–274. DOI:http://dx.doi.org/10.1007/978-3-642-36089-3_15

Igor Böhm, Tobias J. K. Edler von Koch, Stephen C. Kyle, Bjorn Franke, and Nigel Topham. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. New York, 74–85. DOI:http://dx.doi.org/10.1145/1993498.1993508

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, 271–280. DOI:http://dx.doi.org/10.1145/2442516.2442543

Bikin Chiu. 2009. Gmail for Mobile HTML5 Series: Reducing startup latency. Retrieved July 10, 2016, from http://googlecode.blogspot.kr/2009/09/gmail-for-mobile-html5-series-reducing.html.

ECMA International. 2015. ECMAScript 2015 Language specification. *Ecma International*, June, 2015.

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, et al. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, 465–478. DOI:http://dx.doi.org/10.1145/1542476.1542528

Google, Inc. 2016a. Blink. Retrieved July 10, 2016, from http://www.chromium.org/blink.

Google, Inc. 2016b. V8 JavaScript engine. Retrieved July 10, 2016, from https://developers.google.com/v8.

Google, Inc. 2016c. Octane 2.0 JavaScript benchmark. Retrieved July 10, 2016, from https://chromium.github.io/octane.

Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley. 2009. A concurrent trace-based just-in-time compiler for single-threaded JavaScript. In *Proceedings of the Workshop on Parallel Execution of Sequential Programs on Multicore Architectures (PESPMA'09)*.

Hardkernel Co. 2016. ODROID-C1 quad core single board computer. Retrieved July 10, 2016, from http://www.hardkernel.com.

Ian Hickson and David Hyatt. 2011. HTML5: A vocabulary and associated APIs for HTML and XHTML. *W3C Working Draft*, May 25, 2011.

Gary Kacmarcik and Travis Leithead. 2016. UI Events Specification. *W3C Working Draft*, August 4, 2016.

Antti Koivisto. 2008. Implement speculative preloading. Retrieved July 10, 2016, from https://bugs.webkit.org/show_bug.cgi?id=17480.

Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. 2001. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31, 8 (July. 2001), 717–738. DOI:http://dx.doi.org/10.1002/spe.384

LG, Inc. 2016. Open webOS. Retrieved July 10, 2016, from http://www.openwebosproject.org.

Wei Lu, Kenneth Chiu, and Yinfei Pan. 2006. A parallel approach to XML parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID'06)*. IEEE Computer Society, Washington, DC, 223–230. DOI:http://dx.doi.org/10.1109/ICGRID.2006.311019

Chris Marrin. 2011. Webgl specification. Khronos WebGL Working Group.

Mozilla. 2016a. Firefox OS. Retrieved July 10, 2016, from https://www.mozilla.org/en-US/firefox/os.

Mozilla. 2016b. Firefox browser. Retrieved July 10, 2016, from https://www.mozilla.org/en-US/firefox/new.

Mozilla. 2016c. Spidermonkey javascript engine. Retrieved July 10, 2016, from https://developer.mozilla.org/ko/docs/SpiderMonkey.

JinSeok Oh and Soo-Mook Moon. 2015. Snapshot-based loading-time acceleration for web applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. IEEE, 179–189. DOI:http://dx.doi.org/10.1109/CGO.2015.7054198

Addy Osmani. 2015. V8 Optimisations to enable fast page startup. Retrieved July 10, 2016, from https://gist.github.com/addyosmani/671b56d3f69ac4b88f45.

HyukWoo Park, Wonki Jung, and Soo-Mook Moon. 2015. JavaScript ahead-of-time compilation for embedded web platform. In *Proceedings of the 2015 13th IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia'15)*. IEEE, 49–57. DOI:http://dx.doi.org/10.1109/ESTIMedia.2015. 7351768

Filip Pizlo. 2014. Javascriptcore engine. Retrieved July 10, 2016, from https://trac.webkit.org/wiki/JavaScriptCore.

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX, Berkeley, CA, 3–3.

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, 1–12. DOI:http://dx.doi.org/10.1145/1806596.1806598

Seth Thompson. 2015. Custom startup snapshots. Retrieved July 10, 2016, from http://v8project.blogspot.kr/2015/09/custom-startup-snapshots.html.

Tizen. 2016. Tizen platform. Retrieved July 10, 2016, from https://www.tizen.org.

Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. 2013. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Trans. Architect. Code Optimiz. (TACO)* 10, 4, Article 44 (Dec. 2013), 195–226. DOI:http://dx. doi.org/10.1145/2541228.2555301

Received May 2016; revised September 2016; accepted October 2016