공학박사 학위논문

자바스크립트 엔진의 컴파일 과정 최적화에 관한 연구

A Study on Optimization of

Compilation Process for JavaScript Engine

2018년 2월

서울대학교 대학원

전기컴퓨터공학부

박 혁 우

자바스크립트 엔진의
컴파일 과정 최적화에 관한 연구

A Study on Optimization of

Compilation Process for JavaScript Engine

지도 교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함
2017년 11월

서울대학교 대학원
전기컴퓨터공학부
박 혁 우

박혁우의 공학박사 학위논문을 인준함
2017년 12월

위 원 장 이 혁 재 (인)

부위원장 문 수 묵 (인)

위 원 백 윤 흥 (인)

위 원 이 재 진 (인)

위 원 김 수 현 (인)

Abstract

A Study on Optimization of Compilation Process for JavaScript Engine

HyukWoo Park

School of Electrical Engineering and Computer Science

The Graduate School

Seoul National University

JavaScript is a standard dynamic language for web, along with HTML and CSS. Since the appearance of new standards and technologies such as HTML5, ECMAScript6, and WebGL, JavaScript has been increasingly undertaking complex computations, and even running in the server environment. Therefore, the JavaScript performance becomes a critical issue.

For high performant JavaScript, modern JavaScript engines have adopted a multi-tier execution architecture based on the adaptive compilation, which compiles the source code differently based upon the hotness of each function. However, JavaScript engines still suffer from the substantial compilation overhead. This study proposes two optimization techniques for JavaScript compilation process to reduce the compilation overhead, which are concurrent parsing and ahead-of-time compilation (AOTC).

Concurrent parsing approach focuses on the substantial parsing overhead during the web loading time. To reduce the parsing

overhead, concurrent parsing approach performs the parsing process in advance on separate parsing threads, while the main thread directly executes the pre-parsed functions skipping the parsing process. When performed on multi-cores, this can hide the parsing overhead from the main execution thread, thus improving the overall web loading performance. Concurrent parsing technique implemented on a commonly used web browser shows a tangible improvement of the whole web loading performance for various real web apps, which is up to 32% and 18% on average.

AOTC approach is proposed to reduce the whole compilation overhead. From an observation of compute-intensive JavaScript codes, the total compilation overhead accounts for almost half of the entire JavaScript execution time. This result is due to the heavy workload of the parsing and optimizing JITC. To reduce the whole compilation overhead, AOTC technique is employed for JavaScript which stores and reuses the compiled code generated in the previous run. A new AOTC method is discussed which reuses the bytecode and optimized code together for state-of-the-art JavaScript engines. Proposed AOTC method implemented on a commonly used JavaScript engine shows a substantial performance improvement for industry-standard JavaScript benchmark, which is up to 6.36 times and 1.99 times on average.

Two proposed optimizations for JavaScript compilation process efficiently improve the performance of web loading and JavaScript execution with promising results.

Keywords : JavaScript, parsing, JITC, ahead-of-time compilation, compiler optimization, JavaScript engine
Student Number : 2012-30936

Contents

List of Tables

List of Figures

Chapter 1. Introduction

JavaScript is a standard programming language for web along with HTML and CSS. HTML expresses the web components, CSS controls the visual effects, and JavaScript makes the computations, mainly 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.

Since the appearance of new standards and technologies such as HTML5 [1], ECMAScript6 [2], and WebGL [3], the web has been rapidly transformed into a "complete" application platform such as Tizen [4], webOS [5], or Firefox OS [6]. For large-scale web applications (web apps) on these platforms, JavaScript plays an essential role for performing complex computations or user interactivities. Therefore, the JavaScript performance becomes a great concern, 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 [7], just-in-time compilation (JITC) of JavaScript code to machine code [8], or fast initialization of JavaScript built-in objects [9]. Generally, modern JavaScript engines in WebKit [10], Chrome [11] and Firefox [12] include a multi-tier architecture based on the adaptive compilation, which compiles the source code differently based upon the hotness of each function. In the multi-tier architecture, a JavaScript function first goes through the parsing process which parses the source code and generates an intermediate representation, e.g., bytecode. Then interpreter initially executes the bytecode. When the function is continuously executed, baseline JITC translates the bytecode to basic machine code for better performance. When the function is frequently called, then the bytecode is finally re-compiled to highly optimized code

by optimizing JITC. In this way, JavaScript engines efficiently trade off the startup delay and the steady state performance.

However, JavaScript engine still suffers from the substantial compilation overhead. This study proposes two optimization techniques for JavaScript compilation process to reduce the compilation overhead, which are concurrent parsing and ahead-of-time compilation (AOTC).

Concurrent parsing focuses on the performance of web app loading (from the start of web app and until the *load* event fires). From an observation on web loading of real web apps, parsing overhead takes 23% of the whole loading time on average. This result is surprising since web loading generally includes many other jobs such as HTML parsing, CSS rendering, layout, resource downloading as well as JavaScript execution with dynamic compilation. The significant portion of parsing overhead is due to a lot of functions which are called but executed infrequently during the web loading. These cold functions make the parsing overhead substantial other than JITC overhead. Proposed concurrent parsing performs the parsing process in advance on different parsing threads, while the main thread directly executes the parser functions skipping the parsing process. When performed on multi-cores, this can hide the parsing overhead from the main execution thread, thus improving the overall web loading performance.

Although the idea is simple, there are a couple of nontrivial issues involved. First, it is necessary to separate the parsing process from the main thread correctly and make it run on different threads in parallel, without violating the execution semantics of JavaScript. Second, it requires an efficient multi-threaded parsing architecture, which can effectively reduce the synchronization overhead while scheduling the parsing requests promptly. Finally, parsing targets of concurrent parsing should be chosen properly among the functions included in the app, since there will be no benefit if non-called functions are parsed in advance. These issues are discussed specifically in section 3.2.

The contributions of concurrent parsing approach are summarized as follows:

- The bottleneck related to JavaScript parsing in web loading is discovered and concurrent parsing technique is newly proposed to reduce the parsing overhead.
- An efficient multi-threaded parsing architecture is proposed which handles parsing requests in parallel without violating the JavaScript semantics.
- Two methods of choosing parsing targets are explored: one based on profiled information from the previous run, and the other based on speculative heuristics.
- Implementation on a commonly used web browser shows a tangible improvement of the whole web loading performance for various real web apps, which is up to 32% and 18% on average.

AOTC approach is proposed to reduce the whole compilation overhead. From an evaluation of compute-intensive JavaScript codes, the total compilation overhead accounts for almost half of the entire JavaScript execution time, of which parsing and optimizing JITC take considerable portions. Parsing process translates every invoked JavaScript functions to the bytecode with syntactic analysis. Optimizing JITC performs many optimizations during the compilation. Otherwise, baseline JITC merely maps each bytecode to the associate machine code with minimal optimizations, so its overhead is relatively low compared to parsing and optimizing JITC. To relieve the significant overhead of the optimizing JITC, concurrent compilation strategy [13, 14, 15] has been proposed recently. In this technique, the optimizing JITC is handled on separate threads, concurrently with the main thread to hide its overhead. However, this method requires additional CPU cores, and even in the multi-core environment, concurrent compilation operates only for the optimizing JITC thus cannot reduce the total compilation overhead.

To reduce the compilation overhead in a more general way, ahead-of-time compilation approach (AOTC) that stores and reuses the compiled code generated in the previous run is proposed.

In fact, AOTC technique is not a new idea and has been mainly used for Java platforms [16, 17, 18, 19]. In the case of AOTC for JavaScript, it requires different insights and techniques than Java, which are mostly derived from dynamic features of JavaScript. Preliminary works [20, 21] has explored the AOTC for JavaScript, but they simply reuse the bytecode or basic machine code only because they focused on antique JavaScript engines that employ single-tiered architecture without the optimizing JITC. This study suggests a new AOTC technique which reuses the optimized code together for the state-of-the-art JavaScript engines. By reusing the optimized code from the first invocation of the function, the additional performance gain can also be obtained besides reducing the compilation overhead.

Although the idea seems to be simple, several challenging issues are raised when reusing the optimized code. First, speculations applied for the stored optimized code should be validated to check if the code can be reused in the current state. Second, each address value embedded in the optimized code need to be relocated because addresses are changed at each run. Third, some optimizations applied for the optimized code cannot be enabled for AOTC. Finally, the target of AOTC should be chosen among multiple optimized codes.

The contributions of AOTC approach are summarized as follows:

- A novel AOTC approach for modern JavaScript engines is proposed which exploits the bytecode and optimized code simultaneously, considering the performance impact and storage usage.
- The issues related to reusing the optimized code are specified and resolved without severe degradation of the code quality.
- Proposed AOTC implemented on a commonly used JavaScript engine shows a substantial performance improvement for industry-standard JavaScript benchmark and real web apps.

This dissertation is constructed as follows. Chapter 2 introduces the background on web app, JavaScript and modern

JavaScript engines which are essential to understanding the ideas discussed in this study. Chapter 3 describes the concurrent parsing approach. The JavaScript parsing process is first explained in detail. The idea and issues of concurrent JavaScript parsing are discussed sequentially, and evaluation results are given in the end. Chapter 4 describes the JavaScript AOTC approach. Initial AOTC is first introduced, and the proposed AOTC which exploits the optimized code for modern JavaScript engines is discussed. Then, experimental results of AOTC are followed. Related works are described in chapter 5. Finally, chapter 6 discusses the conclusion of this study.

Chapter 2. Background

2.1 Web App

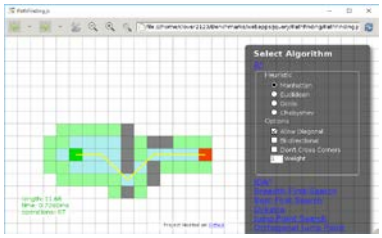
A web app is programmed using HTML, CSS, and JavaScript, as in a regular web page. Figure 2.1 shows an example of a web app, which searches the shortest path between two points using a chosen algorithm. The HTML file includes tags to express the web components, some of which are script tags. Each script tag corresponds to a JavaScript code, often residing in a separate file. Figure 2.1 shows a HTML file including four script tags. JavaScript framework such as jQuery is usually included in the first script tag as in Figure 2.1, 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: web loading followed by event-driven computation. During the web 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 web loading mostly initializes objects including functions and registers them as event handlers. In Figure 2.1, 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 web loading is finished [22]. 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 web loading, a web app works in an event-driven manner

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();
        . . .
});
```

Figure 2.1 Web app example: pathfinding app

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 2.1.

2.2 JavaScript Language

JavaScript is a dynamic-typed scripting language with objects, prototype-based inheritance, first-class functions, and exceptions. Types of variables are not fixed and can be changed during the execution (JavaScript uses keyword *var* instead of type notation for variable declaration). Objects are the fundamental data structure in the language. Object properties are arbitrary strings and can be dynamically inserted into and deleted from objects during the runtime. Functions are also one kind of objects, and can be passed as arguments or return values. Even a JavaScript program can dynamically generate and execute a new code using built-in functions such as *eval* or *Function* with string operations. JavaScript is mainly designed to be elastic even in the face of nonsensical operations such as accessing a non-existent property of an object or adding two functions together; such cases are handled using implicit type conversions and default behaviors in order to continue execution as much as possible without raising an exception.

2.3 JavaScript Engine

JavaScript code in a web app is executed by the JavaScript engine in the browser. Modern JavaScript engines generally adopt multi-tier architecture, which is composed of several execution tiers including interpreter and JITCs, to trade-off the startup delay and the steady state performance. These tiers operate on a function by function basis. Figure 2.2 represents a typical execution process of modern JavaScript engines, such as WebKit's JavaScriptCore [23], Chrome's V8 [24] and Firefox's SpiderMonkey [25]. Since JavaScript is distributed as source code format, JavaScript engine should first translate the source code before executing the code. At the first tier, parser takes this role and translates the source code of an invoked JavaScript function into the bytecode. Then, interpreter initially interprets the bytecode to ensure the quick start of JavaScript execution. When the function is repeatedly called and considered warm, baseline JITC (2nd tier) is triggered to compile the bytecode into basic machine code called baseline code with minimal optimizations. The baseline JITC also inserts instrumentation into the baseline code to collect profile information. The generated baseline code is executed considerable times, e.g., 66 times in JavaScriptCore, to gather stable profile information before transferred to the 3rd tier. When the function is finally found to be hot, its bytecode is re-compiled to the optimized code by the optimizing JITC (3rd tier). Optimizing JITC performs many optimizations especially based on the profile information accumulated during the previous executions.

Since the optimizing JITC performs many optimizations, its overhead is high. To reduce the overhead, modern JavaScript engines also adopt concurrent compilation [13, 14, 15] such that the optimizing JITC is performed on separate threads (compilation threads), concurrently with the main thread to hide its overhead.

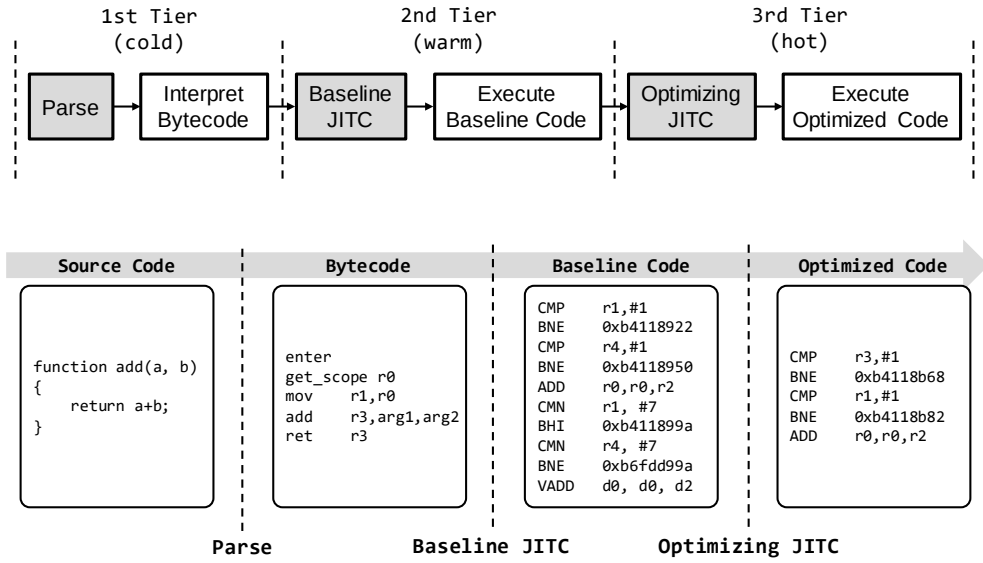


Figure 2.2 Multi-tier execution architecture of JavaScript engine

Chapter 3. Concurrent Parsing[1]

3.1 JavaScript Parsing

This section describes the JavaScript parsing process in detail to understand the key idea of concurrent parsing and provides the motivation of concurrent parsing method, which is parsing overhead.

3.1.1 Parsing Process

As described in section 2.2, JavaScript supports many dynamic features. In addition, JavaScript functions can be declared in a nested form so that a function can be located inside another function. For web apps, JavaScript code is divided based on the script tags. JavaScript parser should deal with these various features.

There are three types of JavaScript code. 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 3.1 (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.

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. No JavaScript rule enforces the lazy

[1] This chapter is part of an article, "Concurrent JavaScript Parsing for Faster Loading of Web Apps," published in the ACM Transactions on Architecture and Code Optimization (Vol 13 Issue 4, December 2016, Article No. 41, DOI: 10.1145/3004281).

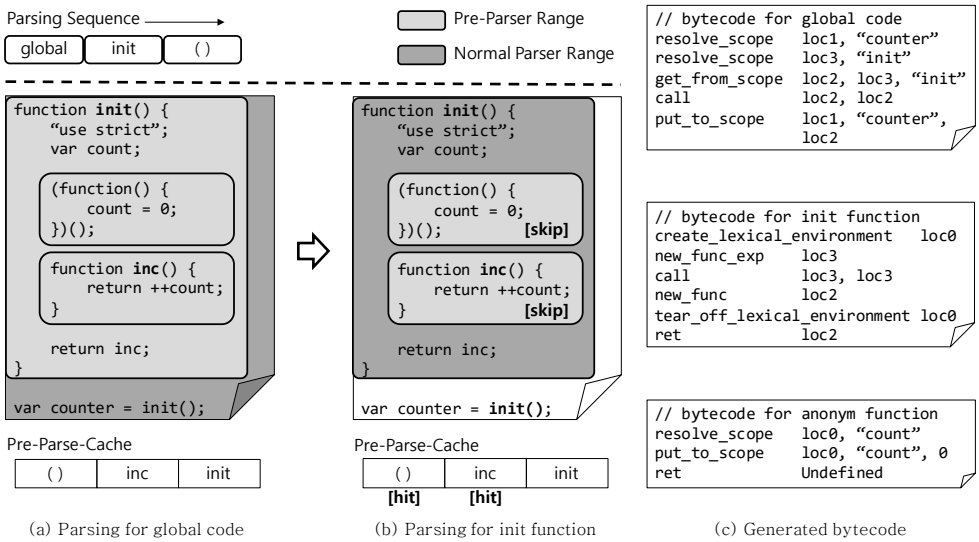


Figure 3.1 JavaScript parsing process

parsing (i.e., it is all right to parse all functions at once), but it is commonly used for the fast startup because many JavaScript functions are not executed during the web loading.

JavaScript allows the nested function 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 names of V8 are used here). 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 example in Figure 3.1, the parsing order will be the same as the execution order. First, JavaScriptCore parses the global code

first by invoking the normal parser, which will generate the bytecode only for the statement var counter = init(); in Figure 3.1 (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, JavaScriptCore will save the information on the anonymous function in a cache called the pre-parse-cache as shown in Figure 3.1 (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 3.1 (c).

Then, when the init function is called by executing the global code, the normal parser will parse init as in Figure 3.1 (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 re-encounter the anonymous function. 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 3.1 (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.

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 the dynamically created code via *eval*. However, there are two parsing semantics that must be respected, 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, the 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 3.1, 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 3.1, 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. Typically, 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, inc can be called by 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 3.1 (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 variables by exploring the lexical environment. If multiple lexical environments are generated, resolve_scope explores them one-by-one to find a matched variable.

3.1.2 Parsing Overhead

Many optimizations have been explored to improve the JavaScript performance. However, one component that has been optimized little, but is getting more important for web apps is JavaScript parsing. Figure 3.2 shows the JavaScript parsing overhead during web loading time (from the start of web app and until the load event fires) for some JavaScript-heavy web apps on the WebKit browser. It is an average of 23% of the whole web loading time, which is surprising since web 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 3.1 represents 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 3.1 (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

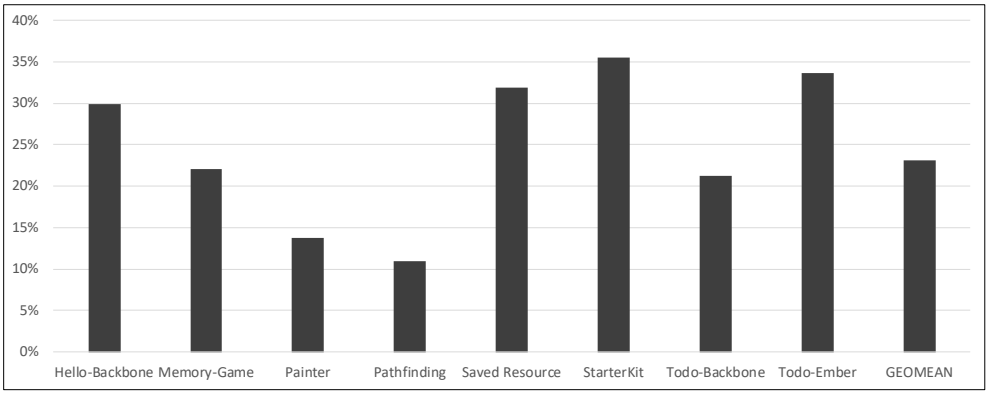


Figure 3.2 JavaScript parsing portion in web loading

Table 3.1 Distributions of JavaScript functions

	Hello-Backbone	Memory Game	Painter	Path finding	Saved Resource	Starter Kit	Todo-Backbone	Todo-Ember
(a) Total	833	794	1522	1458	1075	3636	905	4458
(b)Parsed	155	279	419	328	197	1578	241	1734
(c) Cold	146	218	322	284	190	1380	231	1506
(d) Warm	7	52	67	35	6	148	8	161
(e) Hot	2	9	30	9	1	50	2	67

higher-tier JITC (DFG), respectively. Table 3.1 indicates that many functions are called, but most of them are executed infrequently, not enough to trigger the baseline JITC or optimizing JITC. As a result, JavaScript parsing takes a significant portion of the web loading time. To reduce this substantial parsing overhead, concurrent parsing technique is newly proposed in this study. The details of concurrent parsing will be discussed in continuous sections.

3.2 Concurrent Parsing Approach

3.2.1 Idea of Concurrent Parsing

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 (global code is also regarded as function hereafter). 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 web loading time. 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.

3.2.2 Concurrent Parsing Architecture

Figure 3.3 depicts the overall architecture of concurrent 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, more parsing threads can be added 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 3.4. 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:

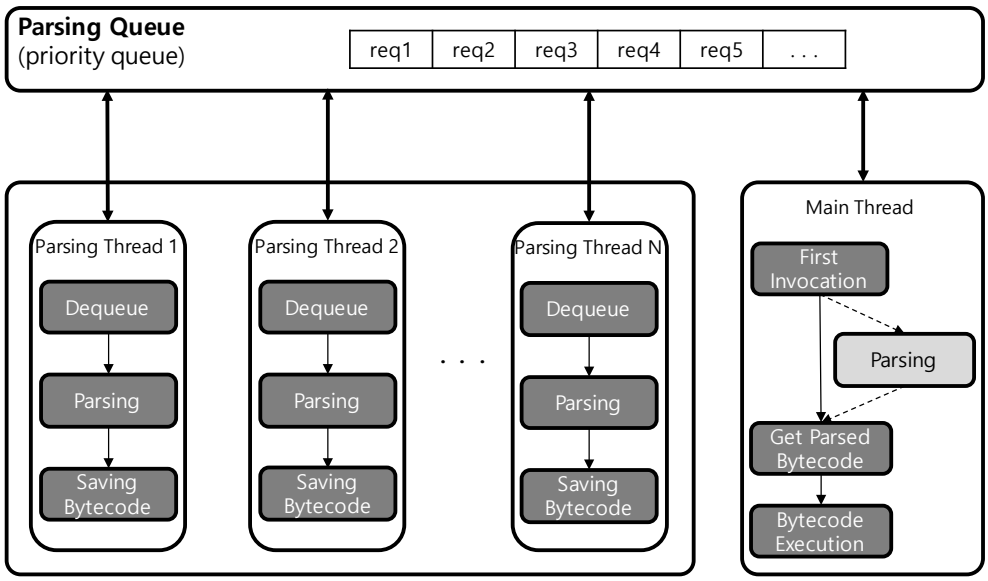


Figure 3.3 Concurrent parsing architecture

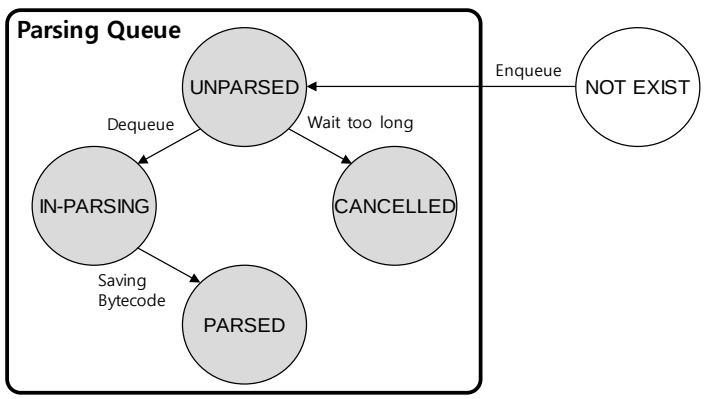


Figure 3.4 State transition of a parsing request

- 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 started parsing for the function yet. In this case, the main thread will immediately cancel it (convert the state to the cancelled state) and parse the function itself 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.

Use of parsing state shared between the main thread and the parsing threads allows synchronization among them with a small overhead. This parsing state is somewhat similar to the compiled state variables used previously for concurrent compilation [14, 15].

3.2.3 Parsing Target Selection

This section discusses how to choose target functions for concurrent parsing. It should be noted that as long as the parsing semantics addressed in section 3.1.1 are respected, 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 functions in an arbitrary order would not parse the functions in a timely way, failing to reduce the parsing overhead. There are 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

Two different approaches will be explored, but both follow these guidelines.

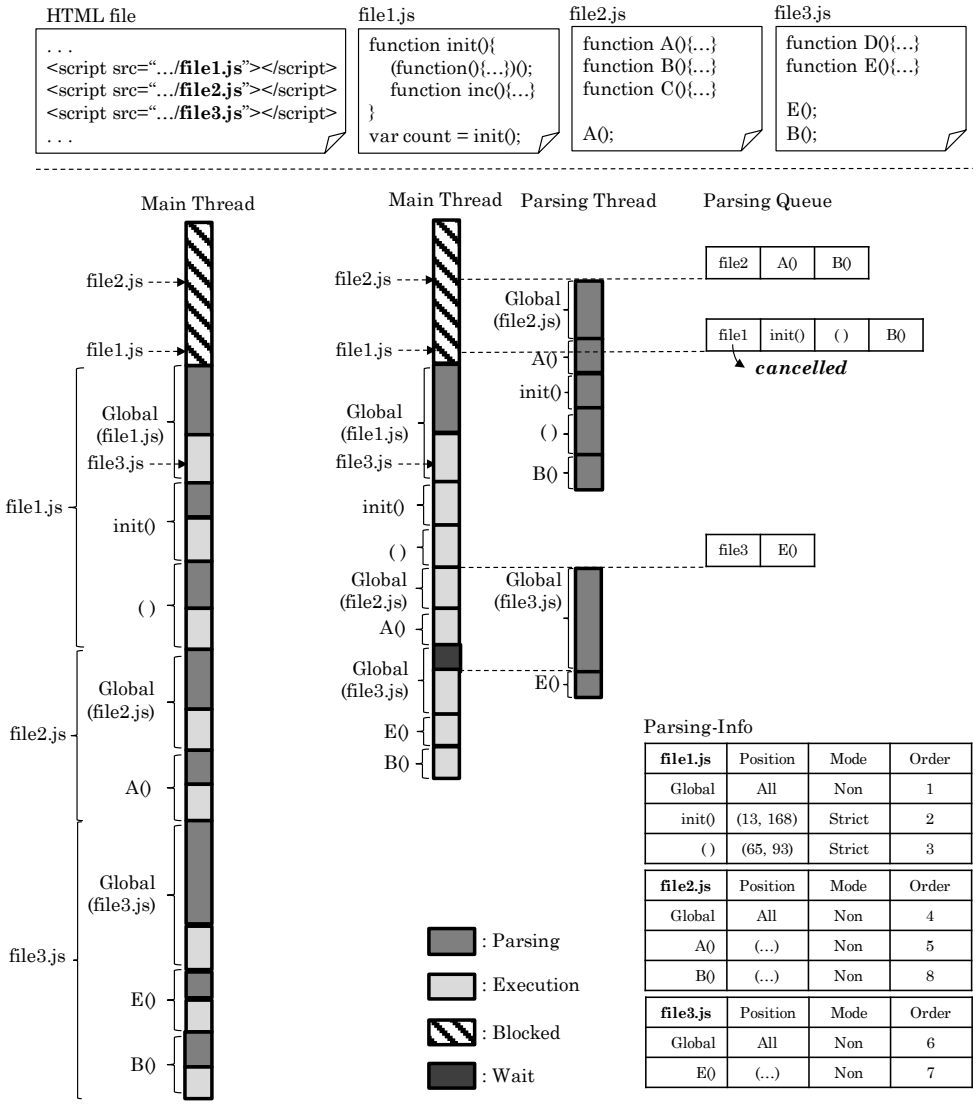
3.2.3.1 Profile-based Concurrent Parsing (PCP)

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 web loading and their first-time execution order in a file,

called parsing-info (so if parsing-info is missing, the browser knows that the web app is executed for the first time). Parsing-info will be used 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 mentioned previously. Fortunately, the same functions are likely to be executed in the same order during web loading, since the web loading tends to repeat the same job, such as the framework initialization or app initialization. So, this approach called PCP, is somewhat ideal case and would show the best performance.

Figure 3.5 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 3.1 (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 3.5 (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 [26, 27]. 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 3.5 (a), fetching for file2 and file3 starts by the pre-loader while file1 is being fetched, so their fetching overhead can be hidden. 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 3.5 (a).

Figure 3.5 (b) shows an example of PCP running with one parsing thread. It also shows the parsing-info for the JavaScript



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

Figure 3.5 Sequential parsing and profile-based concurrent parsing (PCP)

code, which contains the location, the mode (i.e., strict or non-strict), and the execution order for each function executed during web loading. By using the location stored in parsing-info, a parsing thread can identify the parsing range, and by the mode, it can take an appropriate parsing action. Finally, by using the execution order, each request has a priority value and sorted in the parsing queue. Information of closure variables is not recorded in the parsing-info, since detecting the closure variables will be done by the pre-parser

as usual, when the parsing thread handles the normal parsing for the outer functions. In Figure 3.5 (b), parsing-info for file1 has three elements, the global code, the init function, and the anonymous function, which were executed during the web loading. The main thread will read the parsing-info to fill the parsing queue at the beginning of web 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 3.5 (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 the global code of file1, init, the anonymous function, and B in this order as shown in Figure 3.5 (b). This time, however, before the 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 canceled, 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 3.5 (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 3.5 (b) indicates that PCP can

remove much of the original parsing overhead of Figure 3.5 (a), reducing the JavaScript execution time tangibly, if the parsing behavior of the previous run (i.e., parsing-info) repeats in the current run.

3.2.3.2 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. Three simple heuristics are proposed for SCP. One is that those functions located in the preceding script tags should have higher priority than those functions located in following script tags. HTML files usually include multiple JavaScript files as represented in Figure 2.1. These JavaScript files are executed in order from top to bottom. If a function is called from another file, this function should be defined in among the above files. From this feature, functions in preceding files have high priority than functions in following files. 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, only large functions are selected. From an observation, large functions are more likely to be called during web loading than small functions. In this study, 300 byte is used as the threshold value that means functions larger than 300 byte are chosen as the parsing targets in SCP. Functions larger than 300 byte have 47.3% call ratio which is more significant than average call ratio (33%). If a number bigger than 300 byte is used for threshold, SCP can pre-parse necessary functions more accurately, but it may also miss some performance benefit for small functions. So, the moderate threshold value (300 byte) which has nearly 50% call ratio is used.

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 3.1.1, 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 3.6 shows an example of SCP running with one parsing thread, using the same execution scenario of Figure 3.5 (this example assumes 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 3.5, 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 execute it. This is caused by D which is 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 3.6 shows compared to Figure 3.5 (b).

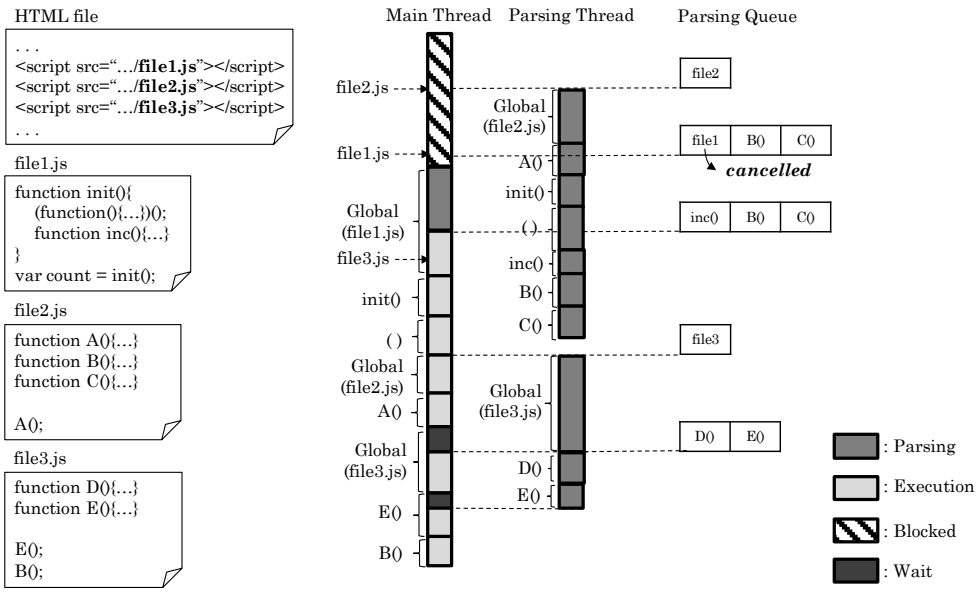


Figure 3.6 Speculation-based concurrent parsing (SCP)

3.3 Implementation on WebKit

Proposed concurrent parsing is implemented on top of the WebKit browser [10] and JavaScriptCore engine [23], a commonly employed platform in Apple's Safari browser [28] or Tizen [4]. Implementation issues related with JavaScriptCore are discussed in this section.

3.3.1 Separation of Parsing Process

In JavaScriptCore, 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 JavaScriptCore engine executes the bytecode via interpretation initially and via JITC as it gets hot. This parsing/execution procedure is depicted in Figure 3.7 (a).

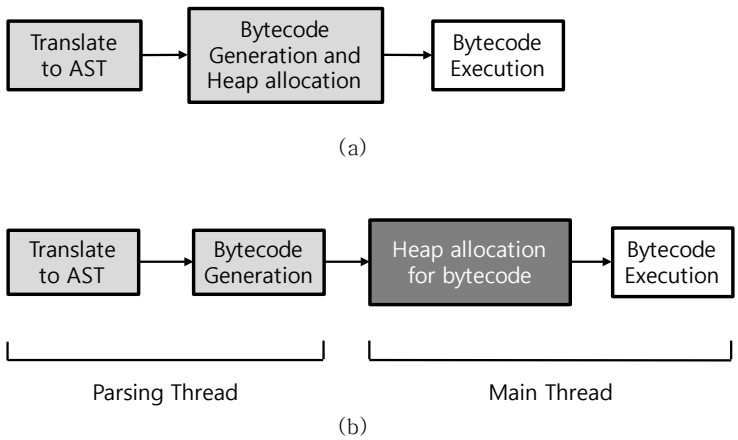


Figure 3.7 Separation of parsing process

In order to separate the parsing job from the main thread to the parsing thread, memory allocation issue related to AST and bytecode should be handled. The JavaScriptCore engine has a single JavaScript heap where all JavaScript objects and their related inner objects are allocated. To avoid the complexities in implementing concurrent memory allocations, every heap allocation is done only by the main thread as usual.

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, bytecode generation in Figure 3.7 (a) is divided 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 3.7 (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 needed. On the other

hand, the bytecode needs to be kept until the main thread takes it away. So a dedicated space for AST is allocated for each parsing thread (and for the main thread as well) and reused for each function parsed in the thread. This technique, known as AST pool allocation [29, 30], has been used for parsing of divided code chunks in parallel, and is also employed here for concurrent parsing. 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.

3.3.2 Shared Data Structure

Two data structures should be shared 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.

JavaScriptCore maintains its own string table for identifiers and constant strings as in Figure 3.8. Each character string is wrapped into a unique StringImpl object, and the address of StringImpl is saved in the string table. JavaScriptCore represents and recognizes each string by StringImpl object internally. Using the string table, JavaScriptCore could prevent duplications of the same string value. When a parser encounters a new identifier during the 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, the string table is shared using the 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 3.1, the pre-parser stores its pre-parsing result in the pre-parse-cache for later reuse. The pre-parse-cache is also shared between the main thread and the

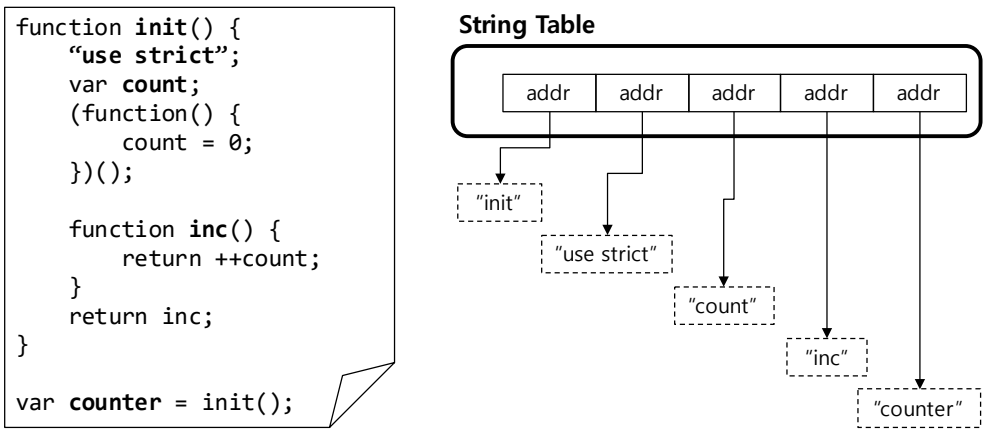


Figure 3.8 String table managed in JavaScriptCore

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.

3.3.3 Source Code Hashing

A web app is generally delivered as source code format from the web server. If the source code of web app is modified by the app developer, browser on client-side could not recognize the change, and PCP approach would be useless, potentially incurring an incorrect execution. Each JavaScript code can be distinguished based on its URL address, but it is not enough to detect the modification of source code. To resolve this issue in PCP, a hashing scheme is employed for each JavaScript code. It caches the calculated hash value of each JavaScript code together with parsing-info. When the system tries to parse each global code concurrently, 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 JavaScriptCore parses that code concurrently. Otherwise, if a modification occurred, JavaScriptCore discards the cached data of the code and runs the parsing process in the main thread as usual. After the loading completes, parsing-info of the modified code is newly recorded.

3.4 Evaluation

3.4.1 Experimental Setup

All experiments run on an ODROID-C1+ ARM board [31], 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). Eight web apps from various domains are experimented, all of which are programmed with one or more JavaScript frameworks, as listed in Table 3.2.

Three types of parsing configurations are evaluated: 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, experiments with different number of parsing threads along with one main thread are done 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 JavaScriptCore engine runs with concurrent JITC threads, so competition among threads will be even higher. In fact, concurrent parsing with two parsing threads achieved the best performance, which will be presented and analyzed shortly.

In this experiment environment, each web app is downloaded and tested 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. Web loading time from the start of web app and until the load event fires is measured. Each experiment measured the loading time ten times and took an average, with the error bound being calculated. For PCP, parsing-info of each app is obtained by running it once in advance. For SCP, the size threshold is predefined as 300 bytes, so only those functions larger than 300 bytes are enqueued to the parsing queue.

Table 3.2 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

3.4.2 Performance Analysis

Figure 3.9 shows the performance improvement of the web 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 most significant parsing overhead, as seen in Figure 3.2. Other apps also show a similar performance gain proportional to their parsing overhead.

Figure 3.10 shows the performance improvement for the JavaScript portion of the web 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 web loading time, and that parsing takes a significant portion of JavaScript execution. Consequently, accelerating JavaScript parsing using the parsing threads as proposed in this study is well justified.

To analyze the performance impact more specifically, the web loading time of the main thread is disassembled as in Figure 3.11. For the original parsing bars, the bottom part shows the parsing overhead of the main thread. For SCP and PCP bars, the bottom part

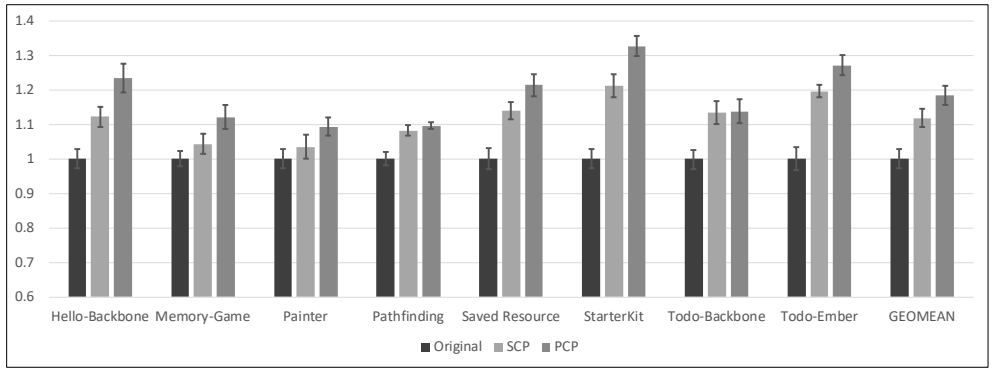


Figure 3.9 Speedup of web loading by SCP and PCP

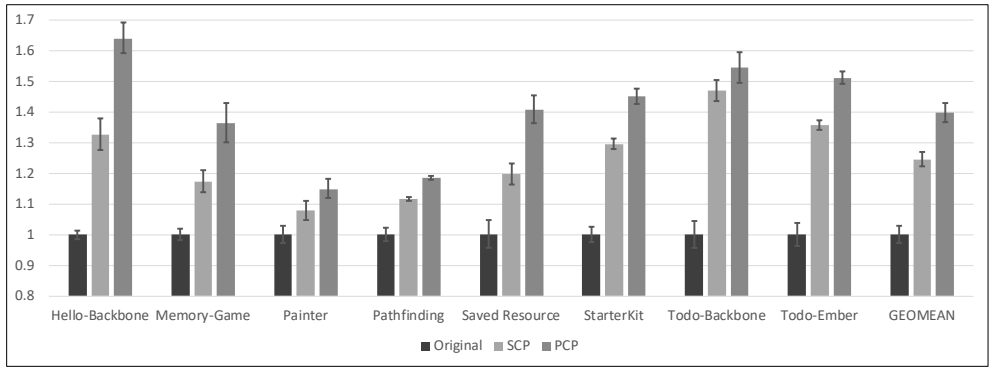


Figure 3.10 Speedup of JavaScript execution by SCP and PCP

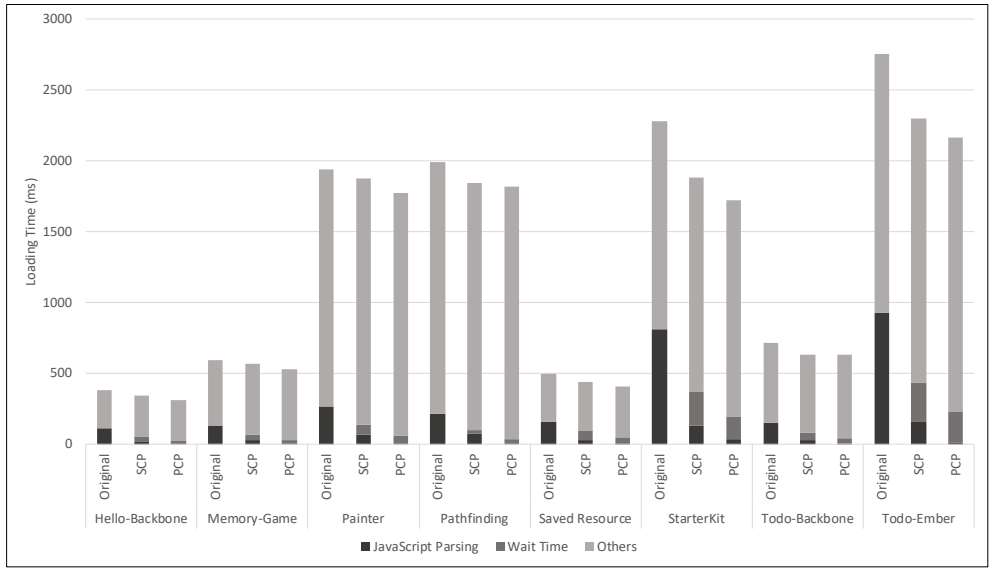


Figure 3.11 Breakdown of web loading time for the main thread

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 3.11 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. The most of this wait time is due to the parsing overhead of the JavaScript framework. Since the size of framework is enormous 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 which 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 3.11, 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 has smaller wait time and parsing time than SCP in Figure 3.11, and these shorter times are the key factor for the better performance of PCP than SCP.

3.4.3 Scalability

Average performance by increasing the number of parsing threads from one to five is measured, as shown in Figure 3.12. Concurrent parsing with two parsing threads achieves the best performance for both SCP and PCP, and the performance goes down as it moves from three to five threads. This is partially due to the overhead of thread creation and synchronization, but 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 [32].

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

Table 3.3 shows the total parsing requests handled by the main thread in each app, for each number of parsing threads in Figure 3.12 (average of ten 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 3.3.

3.4.4 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 3.4 shows the size of parsing-info compared to the size of web app. In Table 3.4, parsing-info size takes almost less than 3% of the web app size, so it is not a significant 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. The hashing overhead affects the loading time negligibly (less than 1%), as in Table 3.5.

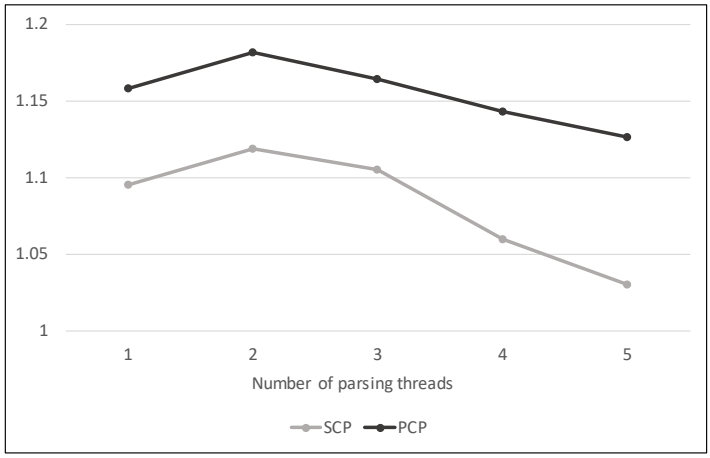


Figure 3.12 Average speedup with different number of parsing threads

Table 3.3 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 3.4 Size of 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
Parsing Info (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	2155.7	371.5	2603.9
Overhead	1.5%	2.2%	3.2%	1.1%	1.4%	2.5%	1.7%	2.4%

Table 3.5 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%

3.4.5 Length of Parsing Queue

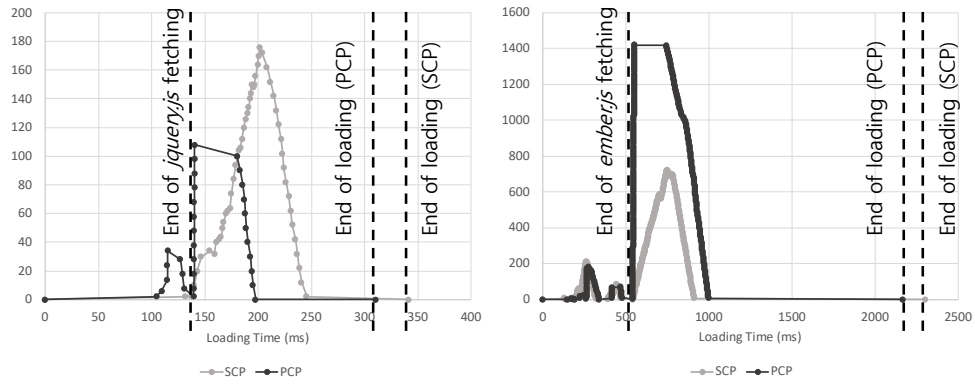
This section examined how the length of the parsing queue changes over the web loading time. Hello-Backbone and Todo-Ember are selected, which have the shortest and longest loading time respectively. Figure 3.13 shows the length of the parsing queue in SCP and PCP with two parsing threads. After the 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.

3.4.6 Extra Experiments

3.4.6.1 JavaScript Benchmark Performance

In addition to the previous experiments on web apps, performance of concurrent parsing on JavaScript benchmark is evaluated too. Figure 3.14 shows the performance of SCP and PCP on Octane benchmark suite [33] with two parsing threads, which is 4.4% and 7.5% on average, respectively. For SCP, every function for concurrent parsing is selected instead of only large size functions, to get a maximum performance because the benchmark behaves differently than web apps. There is tangible speedup for a couple 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 [34, 35], spending more time for



(a) Hello-Backbone (b) Todo-Ember

Figure 3.13 Length of parsing queue over the web loading time

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.

3.4.6.2 Web Page Performance

Concurrent parsing for eight popular web pages from the Alexa list [36] is also evaluated, which have a relatively large JavaScript portion. Unlike the previous app experiment where all source files are located locally at the client, this experiment is performed online, so each web page is loaded from the server and run with concurrent parsing. Figure 3.15 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 3.4.3 and indicates that concurrent parsing would benefit only when there are sufficient cores to run the parsing threads concurrently.

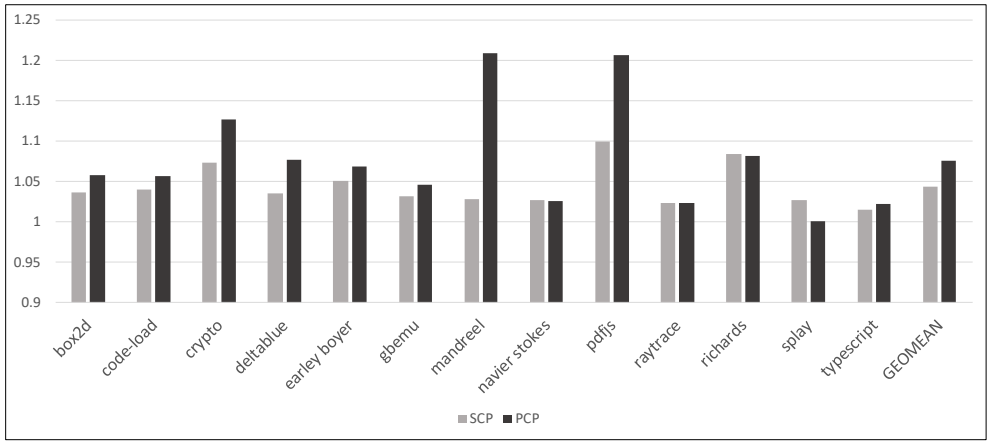


Figure 3.14 Speedup of Octane benchmark by SCP and PCP

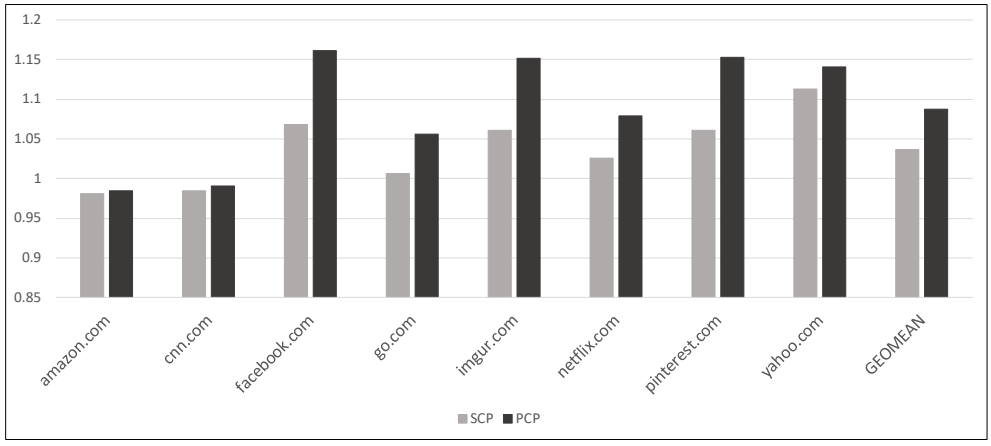


Figure 3.15 Speedup of web page loading by SCP and PCP

Chapter 4. Ahead-of-Time Compilation

4.1 Dynamic Compilation

JavaScript engines execute JavaScript code by dynamic compilation. Parser first translates the source code into the bytecode. JITC dynamically generates the machine code to gain a high performance. Modern JavaScript engines employ two kinds of JIT compilers, which are the baseline JITC and optimizing JITC as depicted in Figure 2.2. This section describes dynamic compilation process, specifically focusing on the optimizing JITC to understand the key idea of the proposed AOTC technique, and provides the motivation of AOTC, which is compilation overhead.

4.1.1 Optimizing JITC

Optimizing JITC performs various optimizations such as common subexpression elimination, loop invariant code motion and function inlining generally used in static compilers. One thing to note is that the optimizing JITC also aggressively applies profile-based optimizations.

The baseline code collects profile information mostly related to the dynamic behaviors such as types of variables or shapes of objects accessed during the execution. Then, optimizing JITC generates the optimized code specialized for the profiled information speculating that the observed behavior will be repeated in the future. Figure 4.1 represents an example of baseline code and optimized code for the same source code. In the source code, global variable glob is initialized with integer value 1. Then, function foo is frequently called in a loop and finally optimized by the optimizing JITC. foo is repeatedly called with an argument object whose x property is initialized as integer value 2. When foo is invoked, it adds two operands which are the x property of the argument object and the global variable glob. In the baseline code of foo, the baseline JITC generates the code of add operation for all operand types to

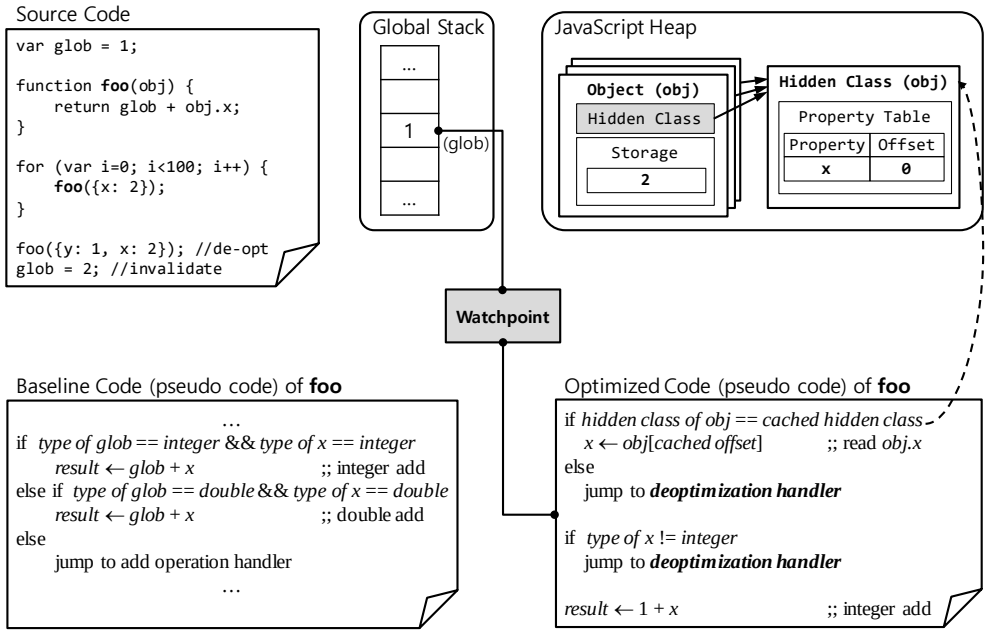


Figure 4.1 An example of baseline code and optimized code for a function

handle the dynamic types.

From the profile information gathered during the baseline code execution, the optimizing JITC recognizes follow things. First, the glob variable is initialized once and remains unchanged. Second, the argument object passed to foo has always the identical shape that property x is located in the first place. Finally, two operands of the add operation, glob and x property of argument, are integer values. Based on this information, the optimizing JITC optimizes the function foo as follows. First, each operand of the add operation needs to be loaded. For the x property of argument, a code that loads the value of property x is generated. In general, JavaScript engines make use of *hidden class*, a list of properties and their offsets in the object, when accessing a property of object. Every object has a reference to the hidden class which represents the object shape. To access a property, the location of the property should first be found from the hidden class. However, foo was observed that it was always called with the identical-shaped object, so the argument object always had one kind of hidden class. Therefore, the optimizing JITC generates the code that directly loads the property x at the observed location in the object without a

reference of the hidden class, expecting that the argument will be the identical-shaped object in the next executions. This optimization of property access is called inline caching [37]. Of course, this speculation may be invalid in the future, e.g., function foo may be called with another-shaped object. To guarantee each speculation, optimizing JITC also inserts guard codes. In this case, the address of observed hidden class is cached and compared to that of the current hidden class of argument object. If two addresses are inconsistent, it means another-shaped object is encountered as represented in the source code of Figure 4.1 and the recovery mechanism called *deoptimization* is triggered. When speculation fails, deoptimization stops the execution at that guard point in the optimized code and resumes at the equivalent point in the baseline code, which is not speculated and hence can handle the execution continuously. In Figure 4.1, *deoptimization handler*, which is a routine defined in the JavaScript engine, is first invoked to handle the switch of execution to the baseline code by recovering the baseline code stack. At this point, JavaScript engine does not discard the optimized code instantly hoping that speculations will be valid in the next executions. After the optimized code is deoptimized several times from the guard code fails, JavaScript engine finally discards the optimized code and re-link the baseline code.

For the other operand glob, the optimizing JITC speculates the glob variable as a constant value because glob is initialized once and unchanged. So, the value of glob (integer 1) is used in the add operation rather than generating the code of accessing and loading the value from the global stack. To guard the speculation on the constant value of glob, *watchpoint* is used instead of guard code. By utilizing the watchpoint, the fail of related speculation is recognized immediately and the guard code is also removed in the optimized code. In Figure 4.1, watchpoint monitors the change of the glob value and has a list of all optimized codes that are speculated on the monitored value. If the glob is modified as in the bottom of source code, the watchpoint is expired directly. Then, its related optimized

codes are abandoned at once, and associated baseline codes are re-linked because the optimized codes are no longer valid. After the invalidation, every invocation of function foo is executed in the baseline code. Each watchpoint is allocated only for specific values which can be accessed globally such as global variables or prototypes because it is inefficient to monitor every value. JavaScript engine manages the watchpoint for later use in the optimizing JITC.

Finally, for the add operation, an add instruction specialized for integer operands is generated based on the profile information. Before the add instruction, guard code that checks the type of property x is inserted. Type check for glob is unnecessary because the glob is already speculated as integer 1 and monitored by the watchpoint.

4.1.2 Compilation Overhead

For high performance, modern JavaScript engines adopt multi-tier execution architecture based on the adaptive compilation, which compiles the source code differently based upon the hotness of each function as described in Figure 2.2. However, JavaScript engines still suffer from the substantial compilation overhead. Since compilation process is dynamically handled during JavaScript execution, its compilation overhead is contained in the runtime. Figure 4.2 shows the overhead of each compilation tier in JavaScript engine measured for Octane benchmark [33]. The total compilation overhead accounts for 52% of the whole JavaScript execution time, of which parsing and optimizing JITC take considerable portions. This result is due to the heavy workload of the parsing and optimizing JITC. Parsing process translates every invoked JavaScript functions to the bytecode with syntactic analysis. Optimizing JITC performs many optimizations, including profile-based optimizations as described in Figure 4.1. On the other hand, baseline JITC merely maps each bytecode to the associate machine code with minimal optimizations, so its overhead is relatively low compared to the parsing and optimizing JITC.

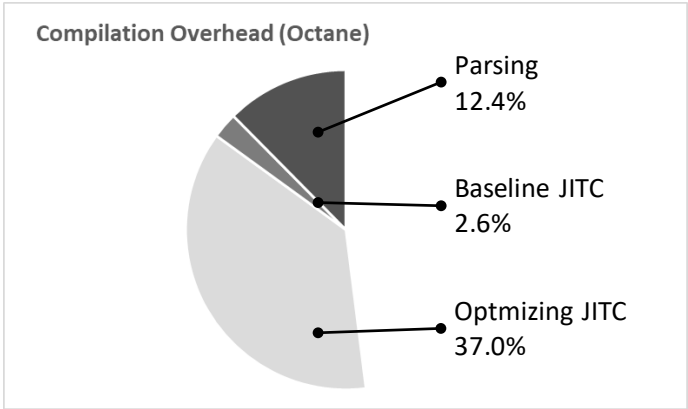


Figure 4.2 Compilation overhead of JavaScript engine during the whole JavaScript execution time, measured for Octane benchmark

To relieve the significant overhead of the optimizing JITC, concurrent compilation strategy [13, 14, 15] has been proposed recently. In this technique, the optimizing JITC is handled on separate threads, concurrently with the main thread to hide its overhead. However, this method requires additional CPU cores, and even in the multi-core environment, concurrent compilation operates only for the optimizing JITC, thus cannot reduce the total compilation overhead.

To reduce the total compilation overhead in a more general way, ahead-of-time compilation (AOTC) approach is newly proposed for JavaScript in this study. The details of JavaScript AOTC will be discussed in continuous sections.

4.2 AOTC Approach

4.2.1 Idea of AOTC

JavaScript engines employ the code cache to keep the code generated by the dynamic compiler, but it is maintained only for the runtime. AOTC technique saves the code generated during the previous run in a persistent cache, e.g., file, and reuses the stored code in the next runs to reduce the total compilation overhead. In fact, AOTC is not a new idea and has been mainly used for Java platforms. JVM-based AOTCs [16, 17, 18, 19] reuse the machine code generated by JITC. The other approaches [38, 39] implement

bytecode-to-C AOTC where the Java bytecode is previously translated to C code which is then compiled by the high-end compiler (gcc) with full optimizations.

This study proposes two AOTC methods for JavaScript environments. In the case of AOTC for JavaScript, it requires different insights and techniques than Java, which are mostly derived from dynamic features of JavaScript. The following section introduces the initial JavaScript AOTC first and then, discusses the main idea of the novel AOTC for modern JavaScript engines.

4.2.2 Initial AOTC[2]

AOTC technique was first proposed and implemented for initial JavaScript engines. Unlike modern JavaScript environments, initial JavaScript engines employ single-tier execution architecture composed of parsing and baseline JITC as represented in Figure 4.3 (this section assumes that WebKit's JavaScriptCore engine on ARM architecture is chosen as the main JavaScript environment, and discusses issues related to it).

Antique JavaScriptCore parses source code into low-level bytecode, which is then directly compiled to the baseline code. The generated baseline code is executed thereafter. The parsing and baseline JITC occur when a function is actually called for the first time. When global code in Figure 4.3 is first executed, JavaScriptCore parses and compiles the global scope (line 1~13), except for the function foo (line 1~12) since foo is not called yet. Then, JavaScriptCore executes the global baseline code which will invoke foo at the point corresponding to the line 13. At this point, JavaScriptCore parses and compiles foo. This lazy compilation strategy also applies to inner functions such as boo located within foo, such that boo will be parsed and compiled when it is called during the execution of foo.

[2] This section is part of an article, "JavaScript Ahead-of-Time Compilation for Embedded Web Platform," published In Proceedings of the Embedded Systems For Real-time Multimedia (2015, DOI: 10.1109/ESTIMedia.2015.7351768).

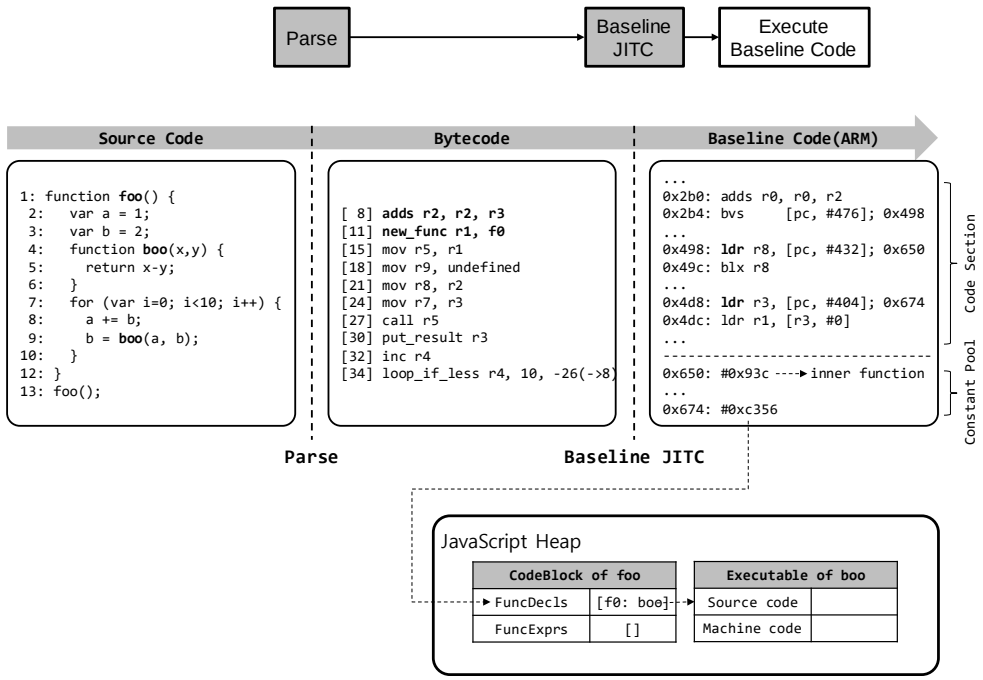


Figure 4.3 Single-tier execution architecture of antique JavaScript engine

Bytecode in Figure 4.3 represents for the loop (line 7-9), which performs the addition and the invocation of the inner function boo. Since it is difficult and burdensome to generate the baseline code precisely and entirely for every bytecode, the baseline code often includes the calls to the inner functions (C functions) defined in the JavaScriptCore. For example in Figure 4.3, the baseline code from 0x2b0 to 0x49c is for the addition bytecode (adds r2, r2, r3). After the add instruction is executed at 0x2b0, there is an overflow check at 0x2b4. If there is an overflow, the baseline code jumps to 0x498 where it loads the address of the inner function and jumps, which will handle the overflow. The address is saved in the baseline code (at 0x650) in the constant pool area, at the end of the code section for the function in ARM.

The baseline code from 0x4d8 to 0x4dc corresponds to the bytecode, new_func r1, f0. When a function is called in JavaScript, a function object is generated which is handled by the new_func bytecode. In Figure 4.3, new_func generates the function object of boo. A function object includes a reference to an Executable object,

which has the source code of the function and the baseline code generated by JITC. To help the generation of this function object during the parsing of the outer scope (i.e., outer function), the Executable object of each inner function is generated and its address is stored in the CodeBlock. CodeBlock object is generated for each function which stores inner function declarations and other information. Figure 4.3 shows the CodeBlock object for the function foo. So the CodeBlock of foo has the address of the Executable of the inner function boo in its FuncDecls array as depicted in Figure 4.3. Now, the baseline code of new_func should access the address of the Executable object stored in the
CodeBlock object to generate a new function object. So, the load instruction at 0x4d8 loads the address of the FuncDecls array stored in the constant pool (which was stored there by baseline JITC). The following load instruction at 0x4dc loads the Executable address of boo from the FuncDecls array.

One crucial issue for AOTC is that the addresses of the inner function and the CodeBlock in the baseline code can change for each run of the app. This means that these addresses should be relocated when AOTC reuses the baseline code. Fortunately, the relocation on ARM can be handled easily because all address values are located in the constant pool only. More details will be discussed below.

4.2.2.1 Initial AOTC Framework

Proposed AOTC system targets web app which is installed as a file format like a smart TV platform, so it can reuse the code in every run, especially web loading because the source code remains unchanged. Since the JavaScriptCore performs parsing and compilation on a function-by-function basis, the unit of AOTC is also a function. AOTC framework stores the bytecode or the baseline code of invoked functions into a file during web loading in the first run of the web app. In the next runs, when a function is called, AOTC framework first checks whether the bytecode or baseline code of that function is stored in the file. If there is none,

JavaScriptCore parses, compiles, and generates the baseline code as usual. If there exists the bytecode of the function, AOTC framework loads the bytecode and generates the baseline code using baseline JITC without the parsing overhead. If the baseline code exists, AOTC loads the baseline code and executes it without the parsing and compilation overhead. Initial AOTC framework implements three methods which reuse the bytecode, the baseline code and both respectively.

4.2.2.2 Reusing the Bytecode

AOTC method which reuses the bytecode can reduce the parsing overhead. Since bytecode is based on virtual register operands, there are no physical address constants in it, so the bytecode itself can be reused as it is. But AOTC framework should consider saving additional data besides the bytecode.

As mentioned in previous, the parser generates the CodeBlock object as well as the bytecode for each function, and the CodeBlock is referenced by the baseline code during the execution. So, AOTC framework needs to save the CodeBlock as well to reuse the bytecode. In addition, the CodeBlock object has the address of the Executable object in the FuncDecls array for each internal function. Therefore, information on the Executable object should also be saved and restored. To save the Executable together with CodeBlock, the number of internal function elements in FuncDecls array is first stored. Then, the offset of the function source code from the beginning of the JavaScript code (not the function source code itself) is saved. Since AOTC assumes the same source code at every run, the offset information is enough to recognize the source code of each function. When restoring the CodeBlock object, each Executable object based on the offset information is generated and its address is allocated in the CodeBlock object.

4.2.2.3 Reusing the Baseline Code

To reuse the baseline code, AOTC framework needs to store three kinds of data: baseline code, CodeBlock, and relocation data. The

case of CodeBlock is handled accurately as in the previous approach.

Reusing the baseline code requires address relocation since the baseline code includes physical addresses. In ARM platform, PC-relative load instructions read address values from the constant pool (CP) area, as seen in Figure 4.3. These addresses would be different at each run, so they should be relocated. For efficient relocation, a new data structure called RelocationInfo is used for relocation of two address types: inner function and CodeBlock.

Baseline code generated by JITC saves the address of inner function in CP to jump to the inner function. Relocation for the inner function is simply replacing the address in CP by the address of the current inner function. For example, Figure 4.4(a) shows the baseline code in the first run. Load instructions at 0x498 access an inner function and its address is stored at 0x650 in CP. An offset of the address from the beginning of the baseline code and its relocation type are saved in RelocationInfo as in Figure 4.4(b). Since the list of inner functions is fixed in JavaScriptCore, a unique ID for each inner function is allocated and saved as the relocation type in the RelocationInfo. In the next-run, the stored baseline code is loaded, and each address is replaced by the current address based on the RelocationInfo as in Figure 4.4(c).

Relocation for the CodeBlock also works similarly by saving information in the RelocationInfo. In Figure 4.4(a), the load instruction at 0x4d8 reads the address at 0x674 which refers to the FuncDecls array in CodeBlock. RelocationInfo has an ID of that FuncDecls as its relocation type. In the next-run, a new CodeBlock is generated, and each address is relocated to the address in the newly created CodeBlock.

Approach of reusing the baseline code can reduce the total compilation overhead, yet the overhead of file I/O operation, restoration of CodeBlock and relocation of address are added. Furthermore, since the size of baseline code is much bigger than

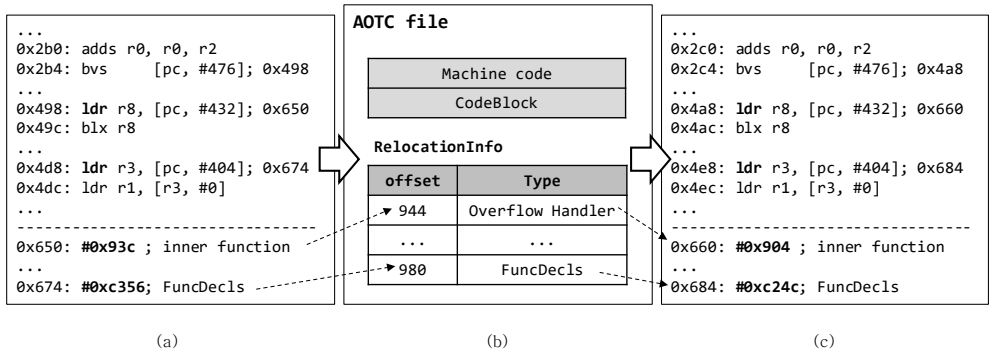


Figure 4.4 Address relocation for baseline code

that of bytecode, this approach consumes enormous storage and the file I/O overhead is also more significant than that of bytecode.

4.2.2.4 Selective Reusing

Reusing the bytecode is advantageous in space usage, while reusing the baseline code achieves a better performance. So, a new approach which selectively reuses the bytecode and the baseline code to maximize each advantage is proposed. This selective method chooses the bytecode or the baseline code for each JavaScript function to get a better performance with small space overhead. The issue is how to select between two codes for a given function. Reusing the baseline code would be effective if the baseline code is small while the baseline JITC overhead is high. So, JITC overhead of each function is measured in the first run and divided by its baseline code size. If this value is larger than a predefined threshold, the baseline code is stored; otherwise, the bytecode is selected. The threshold value is chosen based on profiling and the amount of available storage.

4.2.2.5 Experimental Results

Initial AOTC is evaluated on v8 JavaScript benchmark and four real web apps (Facebook, Financial News, Image Viewer and MTV). v8 benchmark is tested on an ARM board which has an ARM Cortex-A8 1GHz CPU, 256MB RAM, and 4GB storage. Web apps are tested on a smart TV equipped with ARM Cortex-A9 1.2GHz CPU, 1GB RAM, and 4GB available storage. Three AOTC approaches are

evaluated and compared: reusing the bytecode, reusing the baseline code and selective reusing. The threshold value for selective reusing is 0.2 (μs/byte) which is obtained by profiling the compilation overhead and the code size of each function in the profile run. This threshold allows the half of functions to reuse the bytecode and the other half to reuse the baseline code. Execution time is measured ten times and represented as the geometric mean for each experiment.

Figure 4.5 shows the speedup of each AOTC approach. The graph represents the performance of JavaScript execution for v8 benchmark and the web loading performance for web apps. From the figure, all AOTC approaches obtain the performance benefit. The approach of reusing the baseline code gets the highest performance effect which improves the performance by 62% and 27% on average for v8 benchmark and web apps each. Selective reusing shows the middle performance between reusing the baseline and the bytecode. AOTC approaches all achieve more performance gain for web apps than v8 benchmark. This is because, for web apps, repetitive function calls and loop iterations are less than v8 benchmark, which makes the total compilation overhead take a significant portion of the running time

Table 4.1 shows the size of AOTC files compared to the size of JavaScript source code. The approach of reusing the baseline takes the largest space for AOTC file because it stores the large-sized baseline code. Selective reusing takes space of 5.9 times and 4.7 times of source code on average for v8 benchmark and web apps each, which are more closure to that of reusing the bytecode. This result indicates that selective reusing is an efficient compromise because it achieves sufficient performance benefit while consuming moderate space for AOTC files. Selective reusing is affordable especially for embedded platforms which have limited storage space.

This section have introduced the initial AOTC approach for antique JavaScript engines. Based on this approach, this study proposes a new AOTC method for modern JavaScript engines. The following section discusses the main idea of novel AOTC in detail.

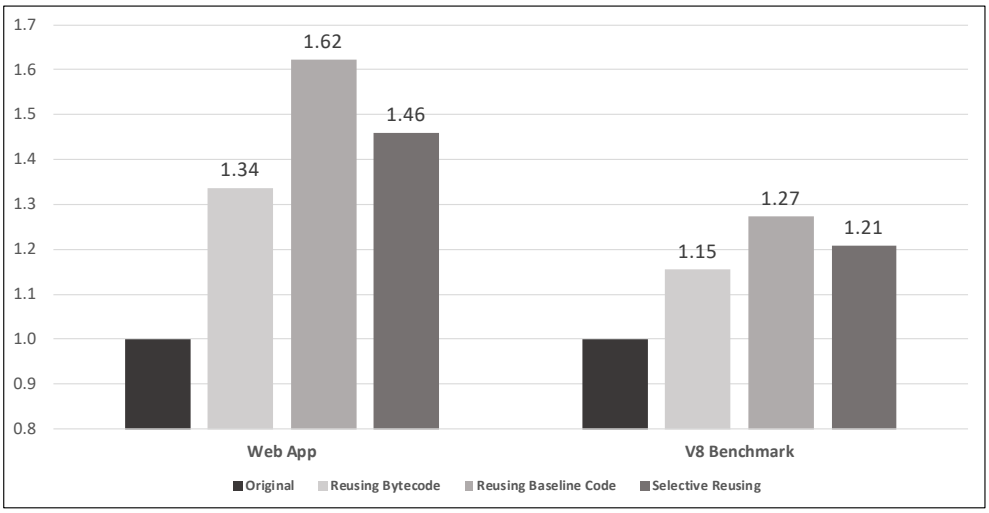


Figure 4.5 Speedup of initial AOTC for v8 benchmark and web apps

Table 4.1 Size of AOTC files compared to the size of source code

Web App	Reusing Bytecode	Reusing Base Code	Selective Reusing
Facebook	**3.1**	**18.6**	**5.8**
Financial News	**1.8**	**11.1**	**3.4**
Image Viewer	**2.1**	**18.0**	**4.8**
MTV	**2.3**	**15.1**	**5.1**
GEOMEAN	**2.3**	**15.4**	**4.7**

V8 Benchmark	Reusing Bytecode	Reusing Base Code	Selective Reusing
Crypto	**2.8**	**17.5**	**5.0**
DeltaBlue	**4.6**	**24.2**	**11.7**
EaleyBoyer	**1.7**	**11.1**	**2.3**
Ray Trace	**3.9**	**23.0**	**6.3**
RegExp	**2.8**	**12.8**	**2.8**
Richards	**4.0**	**19.2**	**17.8**
GEOMEAN	**3.1**	**17.3**	**5.9**

4.3 Novel AOTC

4.3.1 AOTC Framework

For modern JavaScript engines, a new AOTC method, which reuses the bytecode and optimized code simultaneously, is proposed. Since AOTC consumes storage for saving the code, the compromise between the space overhead and the performance impact should be considered. The bytecode, whose size is smaller than machine code, includes the output of the parsing process which is essential for JavaScript execution. The optimized code is the fastest code and has smaller code size than the baseline code (on average, the baseline code is 3.1x larger than the optimized code). Also, the overhead of the parsing and optimizing JITC is substantial while the baseline JITC affects the overall performance negligibly as seen from Figure 4.2. So, both the bytecode and the optimized code are chosen as the target of AOTC excluding the baseline code. When a function is called at the first time, AOTC framework checks whether the compiled code of that function exists in the file. If there is none, the JavaScript engine parses the function and interprets the bytecode as usual. If there exists the code of the function, AOTC framework loads the stored code. For the function which was called only a few times and not optimized by the optimizing JITC in the previous run, only the bytecode of the function was stored in the file. So, framework restores the bytecode and interprets it without the parsing overhead (if the function is continuously executed, the baseline JITC compiles the bytecode as usual). For the function which was frequently executed in the previous run, it loads the stored optimized code and then, directly executes the optimized code as represented in Figure 4.6. In this case, the bytecode is also restored together for two reasons. First, the optimized code needs some JavaScript objects included in the bytecode, such as number or string objects, during the execution. Second, the bytecode is used to generate the baseline code in the case of deoptimization because the proposed AOTC does not reuse the baseline code.

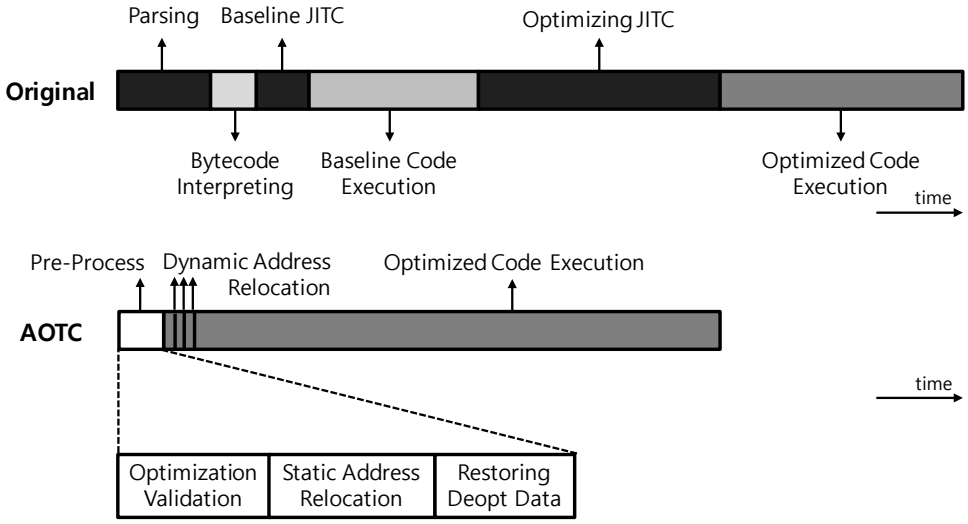


Figure 4.6 Execution sequence of a single function for original and AOTC

Reusing the optimized code can obtain substantial performance benefit by following two aspects.

- The whole compilation process can be skipped including the parsing, baseline JITC and optimizing JITC, thus the compilation overhead is removed completely.
- Early execution of the optimized code from the first invocation of the function rather than going through the front execution tiers.

According to the function-based compilation in JavaScript engine, proposed AOTC reuses the code on a function-by-function basis. Bytecode and its related information are stored in an internal object form of JavaScript engine. To reuse the bytecode, AOTC framework simply saves the bytecode and its related data as a whole, and restores it in the next runs, similar to the method described in section 4.2.2.2. However, reusing the optimized code raises several challenging issues which will be discussed in the next section.

4.3.2 Reusing the Optimized Code

To reuse the optimized code, three pre-process works are required before directly executing the stored optimized code as depicted in Figure 4.6. Restoring the deoptimization data which is one of the pre-process works, is necessary for the deoptimization. Since register-allocated values in the optimized code should be re-allocated in the baseline code stack during the deoptimization, the mapping information of register-allocated values is also stored and restored. Other issues and its solutions will be discussed in detail below.

4.3.2.1 Optimization Validation

First, AOTC framework needs to verify optimizations applied for the stored optimized code to check if the code can be reused in the current state. As mentioned in section 4.1.1, each speculation is guarded by the guard code or watchpoint. For speculations watched by the guard code, AOTC can reuse the guard code as inserted in the optimized code without any validation. However, for speculations guarded by watchpoint, these should be validated because the watchpoint only monitors the modification of the value. For example in Figure 4.1, the glob is speculated as constant value 1 in the optimized code of function foo. When AOTC reuses the optimized code, the glob value may have a different value other than integer 1 in the current state, which it cannot recognize by the watchpoint. To validate the watchpoint-guarded speculations, AOTC saves the monitored value and compares it to the current value. In the case of validation success, AOTC framework links the loaded optimized code to the associated watchpoint to monitor the speculated value. For the case of validation fail, it discards the optimized code and reuses only the bytecode.

4.3.2.2 Address Relocation

The optimized code embeds many address values in itself. Since these addresses are not fixed across each run, they should be relocated to the valid values before reusing the optimized code.

Two addresses are embedded in the optimized code in Figure 4.1, which are hidden class of argument object and deoptimization handler. The deoptimization handler is a static routine which is pre-defined in the JavaScript engine. To handle the address relocation for static addresses such as deoptimization handler, relocation data which includes the location and type of embedded addresses is saved. AOTC framework relocates each embedded static address to the valid address of the current state based on the relocation data. Static address relocation is handled during the pre-process before executing the stored optimized code.

However, the address of hidden class cannot be relocated by the above approach. The hidden class is dynamically allocated in the heap during the runtime. It is hard to find the specific hidden class in the heap, and the hidden class may be not yet allocated when AOTC attempts to reuse the optimized code for the first invocation of the function. To relocate the address of hidden class, the optimizing JITC is modified to generate the optimized code that dynamically re-patches itself as represented in Figure 4.7. The stored optimized code caches an invalid hidden address, so it first jumps to the *property dispatcher* instead of the deoptimization handler. The property dispatcher is also a pre-defined routine which re-patches the cached address and the offset of the property to the valid value by accessing the property of the current object. Then, the next executions of the optimized code can directly access the property if the identical-shaped object is encountered. If another-shaped object is passed in, the property dispatcher is invoked and re-patches the code again. Dynamic address relocation occurs mainly during the first execution of the store code as represented in Figure 4.6. The property dispatcher should be invoked at least once, but it does not degrade the overall performance significantly.

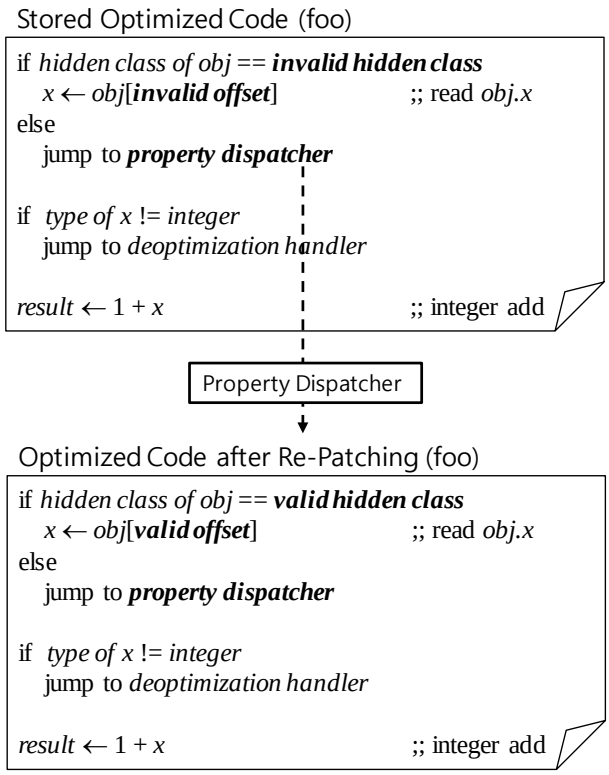


Figure 4.7 Process of dynamic address relocation which dynamically re-patches the optimized code

4.3.2.3 Disabled Optimizations

Despite the previous approaches, two optimizations are hard to be validated in AOTC. First one is function inlining which embeds the body of callee function into the caller function to reduce the function invocation overhead. For JavaScript, each function is also one kind of objects and the address of inlined callee object is cached to check if the current callee function is identical to the inlined function as represented in Figure 4.8. To relocate the address of callee, the method of dynamic re-patching has to compile the body of current callee and re-patches it into the caller code, which is too complicated. Instead of re-patching, validation of inlined function which compares the cached callee and the current callee can be considered. However, it is hard to check if two functions are identical because multiple functions of the same body can be generated during the runtime.

The other tricky optimization is constant value optimization on

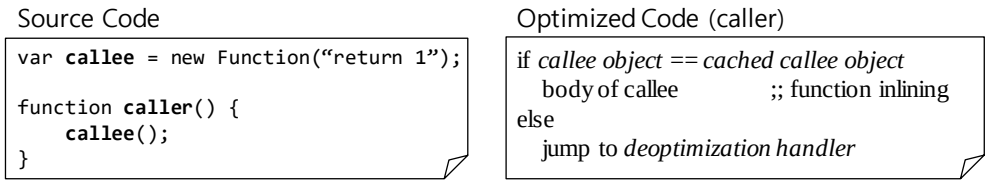


Figure 4.8 An example of function inlining

object values. To verify the optimization, AOTC saves the speculated value and compare it with the current value. However, for the case of object, AOTC should save each property value of the object and compare one by one with the current object, which is too burdensome. Therefore, these optimizations are inevitably disabled for the stored optimized code. Nevertheless, the disabled optimizations decrease the overall JavaScript performance only by 4%, which is endurable.

4.3.2.4 Selection Heuristic

A JavaScript function can be optimized several times by the optimizing JITC. If an optimized code incurs the deoptimization frequently, JavaScript engine regards the code as no longer valid and discards it. Afterward, the baseline code is executed, collecting more profile information. If the baseline code is executed enough, the optimizing JITC re-optimizes it with newly accumulated profile information as represented in Figure 4.9.

If the lastly generated optimized code is reused in AOTC, it will incur fewer deoptimizations than the firstly generated optimized code. But the last optimized code would contain more guard codes due to the further profiled information. Another approach can be considered either which forces re-optimization for every function at the end of JavaScript execution to gather the maximal profile information. However, it is hard to implement this approach because the optimizing JITC uses values in the runtime stack for code optimization. Holding these stack values until the end of JavaScript execution is too burdensome, and even some values may be deleted by GC. So, this study compares only the above two methods which reuse the first or the last optimized code.

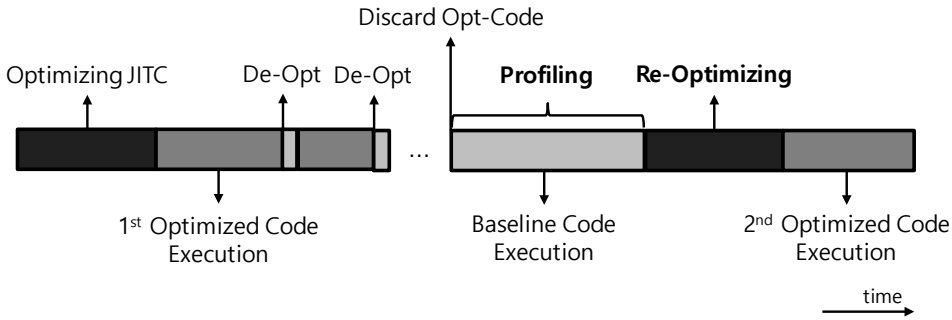


Figure 4.9 Process of re-optimization

4.4 Implementation on WebKit

Proposed AOTC method is implemented on the JavaScriptCore, a commonly employed JavaScript engine in Apple's Safari browser. In this section, several implementation details are discussed.

4.4.1 Address Relocation of String Object

JavaScriptCore internally represents each string value by wrapping the string into a *StringImpl* object. The address of a newly allocated StringImpl object is stored in the string table as depicted in Figure 4.10. JavaScriptCore manages each StringImpl object to contain a unique string value to prevent duplications of the same string values in the memory. Optimized code embeds the addresses of StringImpl objects in itself to recognize the string values. By comparing the address of StringImpl, the optimized code can simply check whether two string values are identical or not. To relocate addresses of StringImpl, another structure called *AOTC string table* is allocate as in Figure 4.10. AOTC string table contains all addresses of StringImpl objects embedded in the optimized code. When saving the optimized code, AOTC framework marks the index of AOTC string table for each embedded address in the relocation data. To reuse the optimized code, AOTC first restores the AOTC string table by allocating the StringImpl objects in the order. Then, it relocates the embedded address based on the marked index number. By using the one shared AOTC string table, AOTC minimally saves the necessary strings rather than saves all character strings for each embedded StringImpl address.

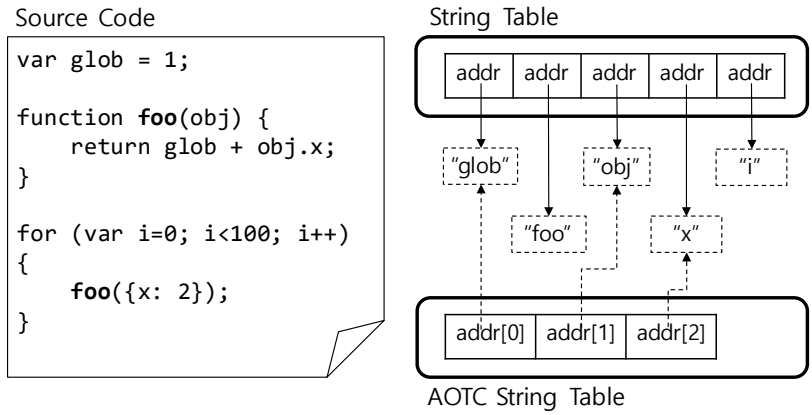


Figure 4.10 An example of string table and AOTC string table

4.4.2 Data Compression

AOTC approach requires storage space for saving the compiled code in a file format. LZ4 [40], a lossless compression algorithm focused on compression speed, is employed to reduce the size of AOTC files. A Data buffer is allocated together to compress/decompress the AOTC data as a whole. When saving the code, every AOTC data is accumulated in the buffer during the execution. At the end of execution, the buffer is compressed once by the compression algorithm and then finally stored in a file. When reusing the stored code, the whole stored data is loaded into the buffer from the file and decompressed. Then, the data buffer is maintained and accessed during the execution when it is necessary. Data compression makes AOTC approach more suitable for space-limited environments, but decompression overhead is also added in every AOTC execution. The overhead of decompression is discussed specifically in section 4.5.

4.4.3 Source Code Hashing

If the source code is modified, AOTC framework cannot recognize it, and thus reusing the stored code would lead to the incorrect execution. To solve this issue, the hash value of each JavaScript code is also stored together with the compiled code similar to the

method described in section 3.3.3. Specifically, a fast hash algorithm [41] is employed to minimize the calculation overhead. When reusing the compiled code, AOTC framework first calculates the hash value of the current source code and checks it with the cached value. If the two values are inconsistent, this means that the source code has been changed. In this case, AOTC framework discards the stored code and runs the modified source code by the multi-tier architecture as usual. After the execution, AOTC framework newly records the compiled code to reuse in later. Otherwise, in the case of consistent, the stored code is directly reused.

4.5 Evaluation

4.5.1 Experimental Setup

Proposed AOTC is evaluated on a quad-core 3.4GHz Intel i7 processor with 16GB RAM, running Ubuntu 15.10. Octane JavaScript benchmark and real web apps were experimented, and the result of Octane benchmark is mainly discussed in this section. All presented results are averages of ten measurements.

Three AOTC configurations are compared together:

- AOTC_BYTE (AB): reusing only the bytecode
- AOTC_ALL_FIRST (AAF): reusing the bytecode and the first optimized code
- AOTC_ALL_LAST (AAL): reusing the bytecode and the last optimized code

To store the compiled code for reusing, each AOTC method ran once in advance.

4.5.2 Performance Analysis

Figure 4.11 represents the performance improvement of Octane benchmark by each AOTC, compared to the performance of original JavaScript engine. The average speedup is 1.15 times (AB), 1.81 times (AAF), and 1.99 times (AAL). All three AOTC approaches improve the JavaScript performance without any performance degradation

AOTC methods of reusing the optimized code achieve much better performance than the method which reuses the bytecode only because they can reduce the whole compilation overhead. To analyze the performance impact more specifically, the JavaScript execution time is disassembled as in Figure 4.12. Figure 4.12 shows that total compilation overhead is generally reduced in AAF and AAL. But the whole compilation overhead cannot be eliminated because compiled codes of some functions refer the extended ASCII codes which are hard to save, so these functions are excluded in AOTC.

Early execution of optimized code can also improve the performance. The performance impact of this aspect is measured by subtracting the reduced compilation overhead from the total performance benefit. As a result, there are significant impacts for a couple of programs such as *box2d* (19.9%), *deltablue* (106.2%) and *navier-stokes* (41.5%). However, for the others, most of the performance effect was obtained by the reduced compilation overhead.

Table 4.2 represents each compilation count and AOTC related numbers for the original and AAL. From Table 4.2, each count of compilation phase is reduced in AAL compared to the original result. Numbers of parsing and optimizing JITC are decreased by reusing the result of each compilation process, which is the bytecode and optimized code. Reusing the optimized code also reduces the number of baseline JITC by skipping the baseline JITC process.

AAL generally shows better performance than AAF. This result is mainly due to the reduced deoptimizations. In Table 4.3, AAL represents smaller or at least equal deoptimization count than that of AAF as expected. AAL reuses the last optimized code which is profiled more and executed mostly in itself with few deoptimizations while AAF employs the first optimized code that incurs frequent deoptimizations due to the insufficient profiling. The performance gap between AAL and AAF indicates that despite the increased type check overhead in the last optimized code, execution entirely handled in the optimized code is critical to the overall performance.

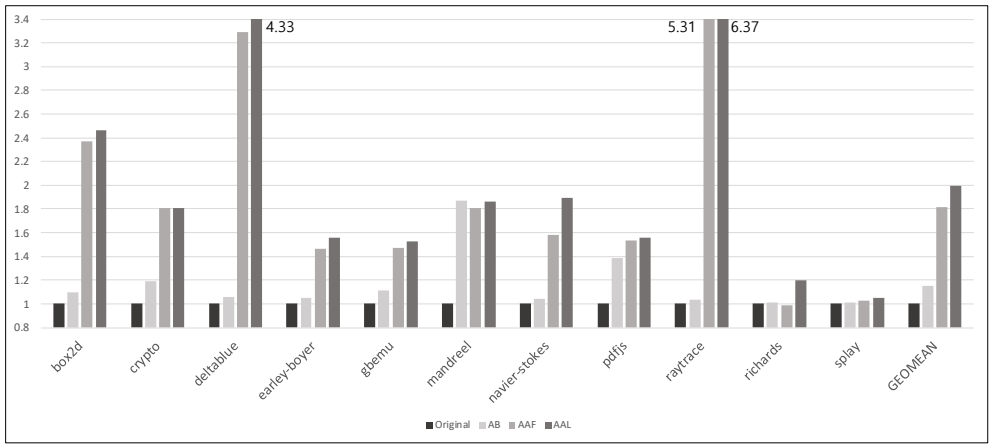


Figure 4.11 Speedup of AOTC for Octane benchmark

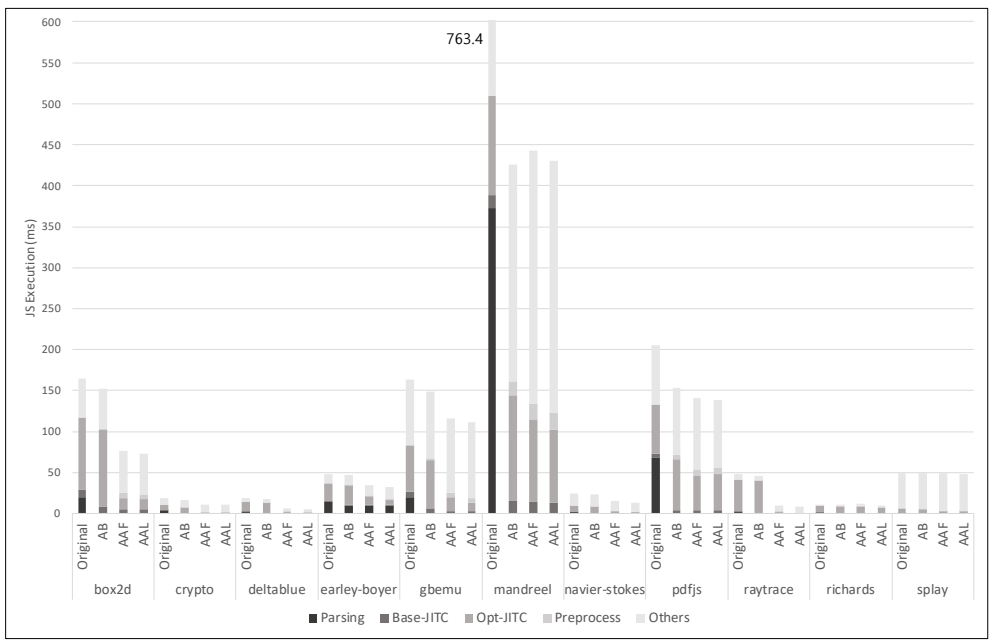


Figure 4.12 Breakdown of JavaScript execution

Table 4.2 Total number of each compilation phase and AOTC related count

Compilation Count		box2d	crypto	deltablue	earley-boyer	gbemu	mandreel	navier-stokes	pdfjs	raytrace	richards	splay
Original	Parsing	332	64	73	88	393	181	29	406	46	34	20
	Base-JITC	213	19	50	36	192	77	10	99	28	17	15
	Opt-JITC	150	10	42	37	183	68	12	78	25	11	8.8
AAL	Parsing	0	0	0	4	1	0	0	9	0	0	0
	Base-JITC	103	11	10	19	49	50	8	62	3	13	8
	Opt-JITC	9	1	1	11	16	38	2	29	0	14	3
	Stored Opt-Code	134	12	41	31	159	69	10	62	26	17	11
	Failed Validation	0	1	0	4	0	36	0	4	0	8	1

Table 4.3 Comparison of deoptimization count

Deopt-Count	box2d	crypto	deltablue	earley-boyer	gbemu	mandreel	navier-stokes	pdfjs	raytrace	richards	splay
Original	7213	42.3	1174	1490	2945	237	11	1793	204	203	20.8
AAF	873	35	1713	2154	1236	1072	11	2906	302	1165	42
AAL	807	35	5	929	242	206	10	1827	0	462	42

From Table 4.3, there are cases that the deoptimization count is increased in AOTC compared to the original. This result is caused by un-profiled execution of 1st tier. Since profile information is gathered only during the execution of baseline code (2nd tier), reusing the optimized code based on this information earlier may incur additional deoptimizations. Otherwise, for cases that the deoptimization count is decreased in AOTC, this is due to the disabled optimizations which induce the reduced speculations and guard codes.

Exceptionally, AAF and AAL obtain similar performance benefits of AB for *mandreel* and *richards*. For both benchmarks, more than the half of stored optimized codes could not be reused because of the failed optimization validation as represented in Table 4.2. These failures are mostly caused by uninitialized or differently initialized global variables. The failure of optimization validation makes reusing the optimized code ineffective, even degrading the performance with substantial preprocess overhead. But this case rarely occurs because the optimization validation targets the watchpoint-based speculations where the watchpoint is allocated for almost constant values, usually initialized by the same value.

4.5.3 Overhead of AOTC

Figure 4.13 shows the size of AOTC files compared to the size of JavaScript source code. By using the compression algorithm, all AOTC methods consume the storage less than three times of the source code size on average, reducing more than the half of the total size.

The decompression overhead is measured to verify if the compression algorithm is feasible. As a result, the decompression overhead is found to be less than 1% of the total execution time, which confirms the compression approach.

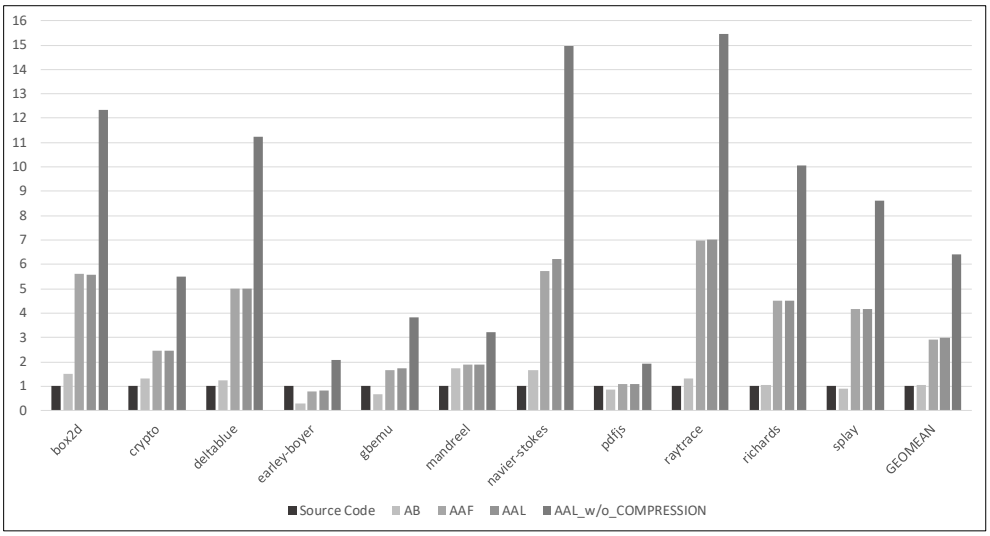


Figure 4.13 Storage usage of each AOTC method compared to the size of source code

4.5.4 Web App Loading Performance

In addition to the previous experiments on Octane benchmark, the performance of web app loading is measured. Web loading represents the time from the start of app and until the first screen of the app is rendered. Since JavaScript code executed during the web loading mostly takes a role of app initialization, it usually repeats the same task at every web loading. Thus, AOTC approach is suitable for the web loading sequence. Figure 4.14 shows the web loading performance of AAL for six JavaScript-heavy web apps, which is speedup of 1.28 times on average. Among them, three web apps show considerable performance improvement by the early execution of optimized code, which are *Painter* (13.5%), *Pathfinding* (13.7%), and *StarterKit* (16.1%). The tested apps all include JavaScript frameworks such as jQuery, and some hot functions are executed for the initialization of these frameworks. By reusing the optimized codes of frameworks, there is substantial speedup for a couple of apps. This result indicates that AOTC approach is promising for real-world apps.

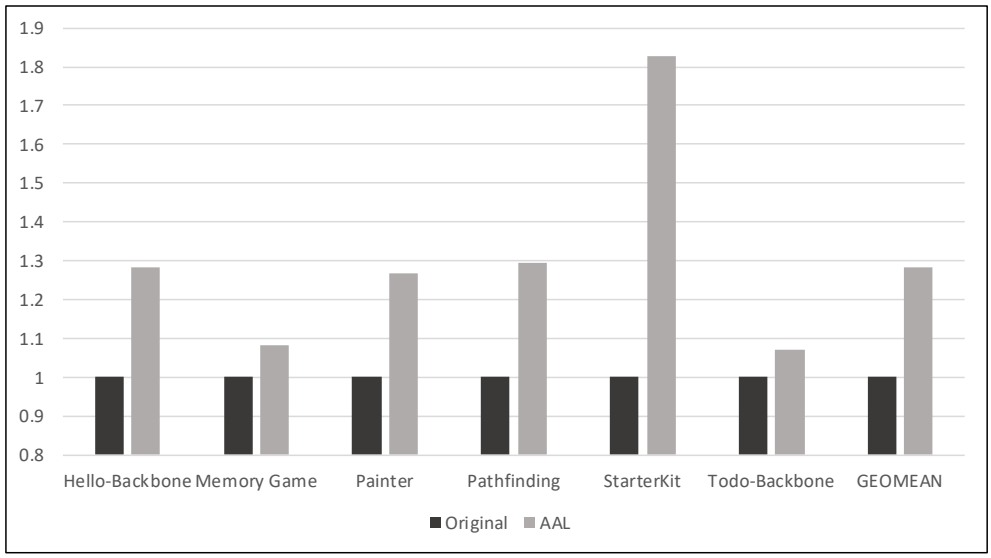


Figure 4.14 Speedup of web app loading

Chapter 5. Related Work

There have been various researches to improve the performance of web loading and JavaScript execution. This section reviews some of that works relevant to this study below.

Chiu [42] described a method to reduce the startup latency by commenting out those JavaScript blocks not needed in web loading in advance, so that they are not parsed during the web loading. To execute those blocks after loading, the app strips out those commented blocks and calls them by *eval*. Proposed concurrent parsing can complement this approach by parsing the uncommented JavaScript code in advance.

HPar [43] proposed parallelizing HTML parsing to minimize the browser's response time. They focused on the HTML parsing and explored two kinds of parallel HTML parsing methods which are pipelining parallelism and data-level parallelism. From the evaluation of real web pages, data-level parallelism which divides the HTML tags into several chunks and parses them in parallel showed tangible performance gain.

Chrome browser recently announced a new technique, script streaming for fast page startup [44]. 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 concurrent parsing in that both parse JavaScript in separate threads for faster loading. But script streaming is only applied to *async* and *defer* scripts based on the expectation that these scrips 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 the DOM tree. For example, web apps tested in this study contain only one async script among entire apps. On the other hand, the concurrent parsing technique 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.

asm.js [45] is a strict subset of JavaScript that is intended to be the target language of translation from statically-typed languages such as C. This language is designed to allow apps written in statically-typed languages to be run on the web browser while maintaining the performance closer to that of native code. Source-to-source compiler such as Emscripten [46] generates asm.js code with type annotations using subtle syntactic hints. This enables the JavaScript engine to optimize the code quickly without online profiling. However, the syntax of asm.js is very restrictive and is not suitable for hand-written JavaScript code. On the other hand, AOTC approach handles the entire JavaScript language without any restriction.

WebAssembly [47] is a new low-level web standard language that provides multiple languages with a compilation target so that they can run on the web, which is similar to the asm.js. Since WebAssembly is a compact binary format with the static type system, it can run faster than JavaScript by initially compiling with existing optimizing JITC. This language is designed to complement and run alongside JavaScript, so it is also an alternative to high performant JavaScript.

Kedlaya et al. [48] described an approach to reduce the number of deoptimizations by using the ahead-of-time profiling (AOTP). The main idea of that paper is to perform offline profiling by running the JavaScript program on the web server. The offline profiler collects information about types and object shapes as well as causes of deoptimizations. Client JavaScript engines then compile hot functions earlier based on the offline profiling information, generating the optimized code to avoid all deoptimizations observed during the offline profiling. AOTP shows 13.1% performance improvement on Octane benchmark (AOTC technique improves the performance of Octane by 1.99 times).

Oh and Moon [49] proposed a snapshot-based technique to accelerate the web loading sequence. They cache snapshots of the

JavaScript heap objects generated during the loading time, and restore the heap from the snapshots to skip the execution of load-time JavaScript code. Although this approach represents substantial performance impact, it is suitable only for the loading sequence. In contrast, proposed AOTC approach can deal with the entire JavaScript execution including the loading time.

Chapter 6. Conclusion

As web pages and web apps increasingly include heavy JavaScript code, the JavaScript performance has become a critical issue. Modern JavaScript engines could achieve a remarkable performance by employing tiered execution architecture based on the interpreter, the baseline JITC, and the optimizing JITC. However, JavaScript engines still suffer from the substantial compilation overhead. To relieve the compilation overhead, this study proposes two optimization techniques, concurrent parsing and AOTC approach.

Concurrent parsing focuses on the parsing overhead during the web loading. To reduce the parsing overhead, concurrent parsing approach performs the parsing process in advance on different parsing threads, while the main thread directly executes the parsed functions skipping the parsing process. Proposed AOTC is implemented on a commonly used web browser and shows a tangible improvement of the whole web loading performance for various real web apps, which is up to 32% and 18% on average.

AOTC approach is proposed to reduce the whole compilation overhead. To reduce the compilation overhead entirely, AOTC approach is employed for JavaScript which stores and reuses the compiled code generated in the previous run. Novel AOTC technique is discussed which exploits the bytecode and optimized code together for state-of-the-art JavaScript engines. Proposed AOTC technique is implemented on a commonly used JavaScript engine and shows a substantial performance improvement for Octane benchmark, which is up to 6.36 times and 1.99 times on average, by reducing the compilation overhead and by running the optimized code from the beginning.

Two proposed optimizations for JavaScript compilation process efficiently improve the JavaScript performance, and are expected to make a more tangible difference for real-world JavaScript-heavy apps in the future.

Bibliography

[1] I. Hickson and D. Hyatt, “HTML5: A vocabulary and associated APIs for HTML and XHTML,” W3C Working Draft, May. 2011.

[2] Ecma International, “ECMAScript 2015 Language Specification,” Ecma International, June. 2015.

[3] C. Marrin, “Webgl specification,” Khronos WebGL Working Group, 2011.

[4] Tizen, “Tizen platform,” July. 2016. [Online]. Available: https://www.tizen.org

[5] LG Inc, “Open webOS,” July. 2016. [Online]. Available: http://www.openwebosproject.org

[6] Mozilla, “Firefox OS,” July. 2016. [Online]. Available: https://www.mozilla.org/en-US/firefox/os

[7] F. Pizlo, “JavaScriptCore engine,” July. 2016. [Online]. Available: https://trac.webkit.org/wiki/ JavaScriptCore

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, “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, NY, 2009, pp. 465-478. DOI: http://dx.doi.org/10.1145/1542476.1542528

[9] S. Thompson. “Custom startup snapshots,” July. 2016. [Online]. Available: http://v8project.blogspot.kr/2015/09/custom-startup-snapshots.html

[10] Apple Inc, “WebKit,” July. 2016. [Online]. Available: https://webkit.org/

[11] Google Inc, “Chrome,” July. 2016. [Online]. Available: https://developer.chrome.com

[12] Mozilla, “Firefox browser,” July. 2016. [Online]. Available:

https://www.mozilla.org/en-US/firefox/new

[13] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder, "Reducing the overhead of dynamic compilation," Software: Practice and Experience, 31, 8, 2001, pp. 717-738. DOI:http://dx.doi.org/10.1002/spe.384

[14] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley, "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), 2009.

[15] I. Böhm, T. J.K. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "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, NY, USA, 2011, pp. 74-85. DOI:http://dx.doi.org/10.1145/1993498.1993508

[16] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta, "Quicksilver: a quasi-static compiler for Java," In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00). ACM, New York, NY, USA, 2000, pp. 66-82. DOI:http://dx. doi.org/10.1145/353171.353176

[17] S. Hong, J.-C. Kim, J. Shin, S.-M. Moon, H.-S. Oh, J. Lee, and H.-K. Choi, "Java client ahead-of-time compiler for embedded systems," In Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '07). ACM, New York, NY, USA, 2007, pp. 63-72. DOI:http://dx. doi.org/10.1145/1254766.1254776

[18] P. G. Joisha, S. P. Midkiff, M. J. Serrano, and M. Gupta, "A framework for efficient reuse of binary code in Java," In Proceedings of the 15th international conference on Supercomputing (ICS '01). ACM, New York, NY, USA, 2001,

pp. 440-453. DOI:http://dx. doi.org/10.1145/377792.377902

[19] L. Zhang and C. Krintz, “Adaptive code unloading for resource-constrained JVMs,” In Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES ’04). ACM, New York, NY, USA, 2004, pp. 155-164. DOI:http://dx. doi.org/10.1145/997163.997186

[20] S. Jeon and J. Choi, “Reuse of JIT compiled code in JavaScript engine,” In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC ’12). New York, NY, USA, 2012, pp. 1840-1842. DOI:http://dx.doi.org/10.1145/2245276.2232075

[21] L. Guckert, M. O'Connor, S. K. Ravindranath, Z. Zhao, and V. J. Reddi, "A case for persistent caching of compiled javascript code in mobile web browsers," In Workshop on AMAS-BT, 2013.

[22] G. Kacmarcik and T. Leithead, “UI Events Specification,” W3C Working Draft, August. 2016.

[23] Apple Inc, “WebKit JavaScriptCore,” July. 2016. [Online]. Available: https://github.com/WebKit/webkit/tree/master/Source/JavaScriptCore

[24] Google Inc, “V8 JavaScript engine,” July. 2016. [Online]. Available: https://developers.google.com/

[25] Mozilla, “SpiderMonkey JavaScript engine,” July. 2016. [Online]. Available: https://developer.mozilla.org/ko/docs/SpiderMonkey

[26] A. Koivisto, “Implement speculative preloading,” July. 2016. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=17480

[27] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar, “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, 2013, pp. 271–280. DOI:http://dx.doi.org/ 10.1145/2442516.2442543

[28] Apple Inc, “Safari browser,” July. 2016. [Online]. Available: http://www.apple.com/safari

[29] W. Lu, K. Chiu, and Y. Pan, ”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, 2006, pp. 223–230. DOI:http://dx.doi.org/10.1109/ICGRID.2006.311019

[30] A. Barenghi, E. Viviani, S. C. Reghizzi, D. Mandrioli, and M. Pradella, “PAPAGENO: a parallel parser generator for operating precedence grammars,” In Proceedings of the 5th International Conference on Software Language Engineering (SLE ’12). Springer, Berlin, Heidelberg, 2012, pp. 264–274. DOI:http://dx.doi.org/10.1007/978–3–642–36089–3_15

[31] Hardkernel Co, “ODROID–C1+ quad core single board computer,” July. 2016. [Online]. Available: http://www.hardkernel.com

[32] Apple Inc, “WebKit2 – High Level Document,” July. 2016. [Online]. Available: https://trac.webkit.org/ wiki/WebKit2

[33] Google Inc, “Octane 2.0 JavaScript benchmark,” July. 2016. [Online]. Available: https://chromium.github.io/octane

[34] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, “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, 2010, pp. 3–3.

[35] G. Richards, S. Lebresne, B. Burg, and J. Vitek, ”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, NY, 2012, pp. 1–12. DOI:http://dx.doi.org/10. 1145/1806596.1806598

[36] Alexa, “The Top 500 Sites on the Web,” July. 2016. [Online].

Available: https://www.alexa.com/topsites

[37] U. Hölzle, C. Chambers, and D. Ungar, “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches,” In Proceedings of the European Conference on Object-Oriented Programming (ECOOP ’91). Springer-Verlag, London, UK, 1991, pp. 21-38.

[38] D.-H. Jung, S.-M. Moon, and S.-H. Bae, “Design and Optimization of a Java Ahead-of-Time Compiler for Embedded Systems,” In Proceedings of the Embedded and Ubiquitous Computing (EUC '08). 2008.

[39] H.-S. Oh, J. H. Yeo, and S.-M. Moon, “Bytecode-to-C ahead-of-time compilation for Android Dalvik virtual machine,” In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE 2015). 2015.

[40] Y. Collet, “LZ4 compression algorithm,” Dec. 2017. [Online]. Available: http://lz4.github.io/lz4/

[41] P. Hsieh, “SuperFastHash,” July. 2016. [Online]. Available: http://www.azillionmonkeys.com/qed/hash.html

[42] Bikin Chiu. 2009. “Gmail for Mobile HTML5 Series: Reducing startup latency,” July. 2016. [Online]. Available: http://googlecode.blogspot.kr/2009/09/gmail-for-mobile-html5-series-reducing.html

[43] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen, “HPar: A practical parallel parser for HTML--taming HTML complexities for parallel parsing,” ACM Transactions on Architecture and Code Optimization (TACO) 10, 4, Article 44, 2013, pp. 195-226. DOI:http://dx.doi.org/10.1145/2541228.2555301

[44] A. Osmani, “V8 Optimisations to enable fast page startup,” July. 2016. [Online]. Available: https://gist.github.com/addyosmani/671b56d3f69ac4b88f45

[45] Mozilla, “asm.js,” Dec. 2017. [Online]. Available: http://asmjs.org/

[46] A. Zakai, “Embscripten: an LLVM-to-JavaScript compiler,” In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (OOPSLA ’11). ACM, New York, NY, USA, 2011, pp. 301-312. DOI:http://dx.doi.org/10.1145/2048147.2048224

[47] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF Bastien, “Bringing the web up to speed with WebAssembly,” In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). ACM, New York, NY, USA, 2017, pp. 185-200. DOI:http://dx.doi.org/10.1145/3062341.3062363

[48] M. N. Kedlaya, B. Robatmili, and B. Hardekopf, “Server-side type profiling for optimizing client-side JavaScript engines,” In Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015). ACM, New York, NY, 2015, pp. 140-153. DOI:http://dx.doi.org/10.1145/2816707.2816719

[49] J. Oh and S.-M. Moon, “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, Washington, DC, 2015, pp. 179-189. DOI:http://dx.doi.org/10.1109/CGO.2015. 7054198

초 록

자바스크립트 언어는 HTML, CSS와 함께 웹 표준 언어를 구성하고 있다. 최근 들어 HTML5, ECMAScript6, 그리고 WebGL과 같은 새로운 표준과 기술들이 등장함에 따라 자바스크립트는 점점 더 복잡한 계산을 수행하게 되었고, 기존의 클라이언트 환경을 벗어나 서버 환경에서도 사용되고 있다. 이처럼 자바스크립트가 많이 활용되는 상황에서 자바스크립트 수행 성능이 중요한 이슈가 되고 있다.

고성능 자바스크립트의 구현을 위해 현대의 자바스크립트 엔진들은 선택형 컴파일 구조(Adaptive Compilation)를 기반으로 한 다단계 수행 모델(Multi-tier Execution Architecture)을 적용하고 있다. 다단계 수행 모델은 자바스크립트 함수의 호출 횟수에 따라 각 함수의 소스 코드를 다른 수준의 최적화를 적용하여 컴파일 하는 방식이다. 하지만 이 방식에서 자바스크립트 엔진은 여전히 상당한 컴파일 오버헤드를 수반하고 있다. 본 연구에서는 자바스크립트 엔진의 컴파일 오버헤드를 개선하기 위해 두 가지 컴파일 과정 최적화 방식을 제안하였는데, 각각 동시 파싱 최적화(Concurrent Parsing)와 선행 컴파일 최적화(AOTC)이다.

동시 파싱 최적화는 웹 로딩 시간에서 상당 부분을 차지하는 파싱 오버헤드에 주목을 하였다. 이 파싱 오버헤드를 개선하기 위해 제안한 동시 파싱 방식은 파싱 과정을 별도의 파싱 쓰레드에서 미리 처리하고, 메인 쓰레드는 미리 파싱된 함수들을 파싱 과정없이 바로 수행하게 된다. 멀티 코어 환경을 이용하면 동시 파싱을 통해 메인 쓰레드에서 파싱 오버헤드를 숨길 수 있고, 전체 웹 로딩 성능을 향상시킬 수 있다. 동시 파싱 방식은 통상적으로 쓰이는 웹 브라우저에 구현되었고, 실제 웹 앱에 대한 실험 결과에서 전체 웹 로딩 성능을 최대 32%, 평균 18% 정도 개선할 수 있었다.

선행 컴파일 최적화는 자바스크립트 엔진의 전체 컴파일 오버헤드를 개선하기 위해 제안되었다. 계산량이 많은 자바스크립트 코드의 수행을 관찰한 결과, 전체 컴파일 오버헤드는 자바스크립트 수행 시간의 절반 가까이를 차지하였다. 이 결과는 파싱과 최적화 적시 컴파일러의 부하가 커서 나온 것이다. 선행 컴파일 방식은 전체 컴파일 오버헤드를 줄이기

위해서 이전 자바스크립트 프로그램 수행에서 컴파일 결과 생성된 코드를 저장하고 재활용한다. 본 연구에서는 현대의 자바스크립트 엔진에 기반하여 최적의 성능을 얻도록 Bytecode와 Optimized code를 재활용 하는 새로운 선행 컴파일 방식을 제안하였다. 선행 컴파일 방식은 통상적으로 쓰이는 자바스크립트 엔진에 구현되었고, 산업 표준으로 쓰이는 자바스크립트 벤치마크에 대한 실험 결과에서 전체 자바스크립트 성능을 최대 6.36배, 평균 1.99배 개선할 수 있었다.

본 연구에서는 자바스크립트 컴파일 과정을 개선하기 위해 두 가지 최적화 방안을 제안하였고, 웹 로딩과 자바스크립트 수행 성능을 개선한 실험 결과를 통해 연구의 가치를 증명하였다.

주요어 : 자바스크립트, 파싱, 적시 컴파일, 선행 컴파일, 컴파일 최적화, 자바스크립트 엔진
학 번 : 2012-30936