<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>An approach to adaptive execution outsourcing in JavaScript intensive web applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Li, Yick Sau Winson (李易修)</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>Li, Y. S. W. (2012). An approach to adaptive execution outsourcing in JavaScript intensive web applications (Outstanding Academic Papers by Students (OAPS)). Retrieved from City University of Hong Kong, CityU Institutional Repository.</td>
</tr>
<tr>
<td><strong>Issue Date</strong></td>
<td>2012</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/2031/6752">http://hdl.handle.net/2031/6752</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>This work is protected by copyright. Reproduction or distribution of the work in any format is prohibited without written permission of the copyright owner. Access is unrestricted.</td>
</tr>
</tbody>
</table>
11CS013

An Approach to Adaptive Execution Outsourcing in JavaScript Intensive Web Applications

(Volume 1 of 1)

Student Name : LI, Yick Sau Winson
Student No. : 
Programme Code : BScCS

Supervisor : Dr CHAN, Wing Kwong Ricky
1st Reader : Dr TAN, Chee Wei
2nd Reader : Dr YU, Yuen Tak
Student Final Year Project Declaration

I have read the project guidelines and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I hereby declare that the work I submitted for my final year project, entitled:

An Approach to Adaptive Execution Outsourcing in JavaScript Intensive Web Applications

does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Project Guidelines.

Student Name: LI, Yick Sau Winson  Signature: __________________________
Student ID: __________________________  Date: __________________________
Abstract

With the advancement of HTML5 and the prevalence of internet enabled mobile devices, the popularity of web applications is increasing. As web applications are becoming more complex, the less powerful JavaScript engines on mobile devices have performance issues when loading computationally intensive web applications. These mobile devices may also have power constraints. Moreover, with the major web browsers each offering its own JavaScript engine, where some are significantly slower than the others, the performance of JavaScript intensive web applications may suffer even on personal computers.

As an attempt to bring the advantages of fast JavaScript engines and powerful computer hardware to these constrained machines, this project proposes a framework for the adaptive execution outsourcing of JavaScript web applications. This framework, named JSCloud, achieves faster execution times on both mobile and desktop devices by migrating selected partitions of the web application code to powerful cloud servers for execution.

In order to prepare a web application to support JSCloud, the web developer only has to annotate methods which are computationally expensive. JSCloud would then inject migration code into the annotated methods. The result is a JSCloud enabled version of the given web application code. This version of code has a cost estimation algorithm which would initiate the migration when the algorithm deems that a reduction in execution time is gained when the execution of the partition is performed on a high performance JavaScript engine in a cloud machine. JSCloud aims to provide devices which have low JavaScript performances with better execution times via execution outsourcing. Experiments have shown that the execution time could be brought down by as much as 50% on devices with slower JavaScript engines.
Acknowledgments

I would like to express my gratitudes to Dr Ricky Chan for his guidance in completing this Final Year Project. Dr Chan has given me useful advices since the conception of the project topic and has provided valuable comments and suggestions on my work throughout the project.

My gratitude extends to Dr Belinda Ho for her lectures on report writing. She has taken the extra effort of interviewing us individually to make sure we are all familiar with report writing.

Ms Dorophy Chu and Mr Bryon Ho deserve a special mention for kindly lending me their mobile phones for testing.
# Table of Contents

1. Introduction ................................................................................................................ 1  
   1.1. Background Information ...................................................................................... 1  
   1.2. Problem Definition ............................................................................................... 3  
   1.3. Aims and Objectives ............................................................................................ 5  
   1.4. Scope .................................................................................................................... 5  
   1.5. Deliverables .......................................................................................................... 6  
   1.6. Overview of the Report ........................................................................................ 6  

2. Literature Review ....................................................................................................... 7  
   2.1. Web Application Overview .................................................................................. 7  
   2.2. Client-Server Communication .............................................................................. 9  
   2.3. Augmented Computation ...................................................................................... 10  
   2.4. Code Partitioning .................................................................................................. 10  
   2.5. Migration and Re-integration ............................................................................... 11  
   2.6. Migration Trigger ................................................................................................. 11  
   2.7. Comparing JSCloud with Existing Efforts ........................................................... 12  

3. System Design ............................................................................................................ 13  
   3.1. System Overview ................................................................................................. 13  
   3.2. Use Case Modeling .............................................................................................. 15  
   3.3. Object Modeling ................................................................................................... 19  
   3.4. High-Level Object Interaction Modeling ............................................................. 20 

4. Methodology and Implementation ............................................................................. 22  
   4.1. JavaScript Code Modifier .................................................................................... 22  
      4.1.1. HTTP Proxy ............................................................................................. 22  
      4.1.2. Code Analyzer .......................................................................................... 24  
      4.1.3. Code Modifier .......................................................................................... 26  
   4.2. Modified JavaScript File ...................................................................................... 26  
      4.2.1. Cost Estimator .......................................................................................... 27
1. Introduction

1.1. Background Information

In recent years, web applications have gained popularity. Client side web applications are applications executed on web browsers such as Internet Explorer [1] and Safari [2]. They can be conveniently accessed over the internet, requiring no installation and are cross platform. The only requirement is a standard compliant web browser.

Among the most prevalent web applications are social networking services and collective intelligence services. Examples include Gmail [3], Facebook [4], Twitter [5] and Wikipedia [6]. These form one of the key characteristics of the current internet age. Another common type of web applications are Software as a Service (SaaS) [7]. Also known as on-demand software, SaaS is a model where software and user data are stored in a centralized location and are accessed from web browsers. Google Documents is an example. In other words, desktop applications are being moved onto the cloud. Web applications seem to have became increasingly powerful and yet complex.

The availability of mobile devices have made web applications even more accessible. However, due to the limited hardware performance and screen size, some web applications could be impractical for use on mobile devices in terms of performance and interface usability. As such, many mobile versions of web applications with optimized interfaces have been released (for example the Facebook for Mobile webpage [8]). Native applications for specific mobile platforms, which are commonly known as “apps”, are also developed to provide an optimized functionality for mobile devices (such as Facebook for Mobile [9]). However, such optimizations comes at the cost of reduced capabilities. Most mobile versions offer only a subset of features provided by the standard counterpart (for example, the lab features of Gmail [10] are not available on the Gmail mobile app).
Usability limitations due to small screen sizes could be mitigated by tablet devices which offer bigger screens. Yet, the technical limitations due to processing power remains. The computation of certain code could be taking too long on mobile devices.

Cloud servers have higher performance than mobile devices and personal computers (PC). It would be possible to harness the computing power of cloud servers for computation tasks. This project would explore the possibilities of offloading complex computations in the client machines to cloud servers as an attempt to reduce processing time.
1.2. Problem Definition

To illustrate the problem, we use the example of a spreadsheet web application. A column of integers modeled as array `col` of length `col.length` would be sorted. The computation time of the following merge sort algorithm implemented in JavaScript to sort `col` would be measured. The results are shown in Table 1.1.

```javascript
function merge_sort(arr)
{
    function split_array(arr)
    {
        if (arr.length <= 1)
            return arr;

        var middle = parseInt(arr.length / 2);
        var left = arr.slice(0, middle);
        var right = arr.slice(middle, arr.length);

        return merge(split_array(left), split_array(right));
    }

    function merge(left, right)
    {
        var result = [];
        while (left.length > 0 || right.length > 0)
        {
            if (left.length > 0 && right.length > 0)
            {
                if (left[0] <= right[0])
                    result.push(left.shift());
                else
                    result.push(right.shift());
            }
            else if (left.length > 0)
            {
                result.push(left.shift());
            }
            else if (right.length > 0)
            {
                result.push(right.shift());
            }
        }

        return result;
    }

    return split_array(arr);
}

// timestamp t1 in milliseconds
merge_sort(col);
// timestamp t2 in milliseconds
```

Figure 1.1 Algorithm of `merge_sort`
<table>
<thead>
<tr>
<th>(a) Laptop computer</th>
<th>MacBook Pro 2.2Ghz C2D Chrome v.14</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>21</th>
<th>773</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Laptop computer</td>
<td>MacBook Pro 2.2Ghz C2D Safari v.5 (Default browser)</td>
<td>0</td>
<td>1</td>
<td>28</td>
<td>1531</td>
<td>-</td>
</tr>
<tr>
<td>(c) Laptop computer</td>
<td>Lenovo X200 2.26Ghz C2D IE 9 (Default browser)</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>135</td>
<td>6561</td>
</tr>
<tr>
<td>(d) Mobile phone</td>
<td>BlackBerry 9780 Default browser</td>
<td>2</td>
<td>11</td>
<td>261</td>
<td>18827</td>
<td>-</td>
</tr>
<tr>
<td>(e) Portable Multimedia Player</td>
<td>iPod Touch 1st Generation Safari (Default browser)</td>
<td>7</td>
<td>22</td>
<td>420</td>
<td>26167</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1.1 Execution time of merge_sort() for different list sizes on different platforms.

Through this experiment we have identified the following problems concerning web application response times.

Operations which are responsive on PCs can be slow on mobile devices. In the case of sorting 1,000 numbers, we observe from Table 1.1 that the computation time on mobile devices (d) and (e) exceed 100 ms. At this level of delay, the user would notice a delay in the response [11, 12].

Performance of can be low with some browsers even on the same PC. Considering (a) and (b), which are different browsers on the same system, we see that the merge_sort took 21 ms to complete in Chrome while it exceeded 1000 ms with Safari.

Very large computations can lead to the script fail to complete on both mobiles devices and PCs. When sorting 100,000 numbers, the script failed to complete in PC (b), mobile devices (d) and (e). However in (a), it completes successfully within 773 ms. On desktop machines, the browser issues a warning and prompts the user if he wants to terminate the script. On mobile devices, the browser would become unresponsive and would require a force quit.
1.3. Aims and Objectives

This project aims to propose a solution to mitigate the limitations incurred on web applications due to the limited processing powers of mobile devices. Taking inspiration from the CloneCloud [13] project, this project would clone client side application code to cloud servers for computation. Limitations of this model would be addressed followed by some discussion. For the ease of illustration, the system to be developed is named JSCloud. JSCloud aims to provide shortened execution time of web applications through migrated execution. JSCloud would transparently and automatically modify the web application code to perform the necessary migration procedure.

1.4. Scope

This project would focus on the client-side program of web applications implemented in JavaScript. JavaScript [14] is a scripting language for computation and interaction with the Document Object Model (DOM) of webpages. The reason for focusing on JavaScript is primarily for its prevalence and support for cross platform compatibility. Each JavaScript program runs on a JavaScript Virtual Machines (VM), which are also known as JavaScript engines. They are built into standard compliant browsers and require no installation of additional plugins. As a result, JavaScript web applications are compatible with all computers and mobile devices which have standard compliant web browsers.

The logic for migration of JavaScript code partitions to the cloud server could be performed in the JavaScript VM or in a module written as a JavaScript class. This project would implement the cloning logic as a JavaScript module. The justification would be provided in the Section 3.

Some optimizations to JSCloud would be made to address a subset of problems. Considerations like security issues would not be in the scope of this project. In short, this project provides a working prototype of JSCloud only.
1.5. Deliverables

Apart from the delivery of the JSCloud module, supporting documents would be produced.

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Target Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Plan</td>
<td>24-Oct-11</td>
</tr>
<tr>
<td>Interim Report</td>
<td>5-Feb-12</td>
</tr>
<tr>
<td>System Demonstration</td>
<td>10-Apr-12</td>
</tr>
<tr>
<td>Final Report</td>
<td>10-Apr-12</td>
</tr>
</tbody>
</table>

1.6. Overview of the Report

The rest of the report is divided into five sections. Section 2 would cover the Literature Review. Section 3 is the documentation of the System Design. Section 4 describes the Methodology and Implementation. Section 5 presents the Performance Evaluation. Section 6 is the Conclusion.
2. Literature Review

2.1. Web Application Overview

A typical web application consists of programs running on both the client $C$ and server $S$. The computation of a web application begins with a client-side application $P$ being downloaded to the web browser from a HyperText Transfer Protocol (HTTP) server. $P$ provides the user interface to the user. When resources in a web server is required by $P$, $P$ sends a request to the HTTP server that can reply the request with a response. Figure 2.1 shows the overview of a typical web application.

![Diagram illustrating the relation among user-client-server in a typical web application.](image)

The client-side application is transferred as HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS) files. Each HTML file describes some contents and structure of the webpage. Each CSS file specifies and controls the presentation style of the webpage. JavaScript is the language used in client side application programming. JavaScript manipulates objects in the Document Object Model (DOM) [15] of a web browser. The DOM is an interface defining the logical document structure and the methods to access and manipulate documents. The DOM object tree contains HTML documents, CSS and JS source scripts. The webpage (i.e. user interface) is a rendering of the DOM object tree by the web browser layout engine. Thus, manipulating the DOM objects would manipulate the UI. JavaScript manipulates and accesses the DOM objects through the DOM application programming interface (API). Figure 2.2 shows the relation between the DOM, UI and JS.
To further illustrate, the spreadsheet sorting web application used as example in Section 1.2 would have the structure shown in Figure 2.3. The `addEventListener()`, `getElementById()` and `setAttribute()` methods invoked by JavaScript are provided by the DOM. The DOM would listen for the left click event on the sort button on the webpage to invoke `sortCol()`. The sorted column would then be displayed to the spreadsheet by updating each node of the spreadsheet with the sorted array. Updates to the DOM tree (i.e. the Object Tree in Figure 2.3) would automatically be rendered to the UI in accordance to the DOM standard.

**Figure 2.3** Example of how JavaScript manipulates DOM Objects in a web application.
2.2. Client-Server Communication

Asynchronous JavaScript and XML (AJAX) [16] is one of the fundamentals of the modern web applications. It is a development method that enables asynchronous requests between the server and the client. The client can perform other operations while waiting for the responses from the server. Despite the name XML, data transferred by AJAX could also be in JavaScript Object Notion (JSON) [17], plaintext, or any data model. AJAX disassociates data request from HTML page requests. With AJAX, full webpage reloads are not necessary for new data to be transferred. Requests are made in the background and the user stays on the same page. Upon receiving the reply from the server, a JavaScript callback function would be executed to show the information to the user by manipulating the DOM which in turn causes the UI of the webpage to be re-rendered. AJAX requests are made by the XMLHttpRequest method.

To convert JavaScript objects to JSON format, an external library json2.js [18] is used. The library contains two methods. The `stringify` method serializes the object passed into it and returns the object in JSON representation. The method simply casts the object passed into it to a string to produce the JSON string. The `parse` method deserializes the JSON string passed into it and returns the deserialized object. The parsing of the JSON string is performed in three stages. First, all unicode characters are converted to escape sequences. Then, the JSON string is validated by applying multiple regular expression matches. An error is thrown if the JSON string is invalid. In the last stage, the objects serialized in the JSON string are recursively deserialized by the JavaScript built-in `eval` method. The `eval` method executes the string passed into it as JavaScript code (which is similar to the backtick ` operator in the bash shell). For example, `eval ("[‘foo’, ‘bar’]")` returns the array `[‘foo’, ‘bar’].`
2.3. Augmented Computation

Augmented computation refers to the execution of certain partitions of a program on a remote machine. In the scope of this project, the purpose is to migrate the computation of complicated code from resource-starved devices to more powerful machines in order to gain higher performance.

![Illustration of augmented computation. Code partitions represented as circles.](image)

A working prototype of augmented computation on mobile devices by migrating code to cloud servers for high performance computation called CloneCloud [13] has been experimented by Chun et. al, allowing up to 20 faster lower computation speed.

2.3.1. Code Partitioning

Augmented computation begins with code partitioning - to divide the code into partitions which are suitable for migration and remote computation. This process can be carried out at various levels. Chun et. al partitioned code at the VM level [13], while Gu et. al, Messer et. al and Ou et. al partitioned at the Java class level [19, 20, 21]. Partitioning at different levels calls for different considerations and techniques. In this section, we would review the principles for partitioning CloneCloud used.
As code can be partitioned at any point in theory, practical implementation calls for specific partition points. CloneCloud restricts partition points to method entry and exit points. Additionally, partitions have to fulfill a set of properties to become legal. The first requirement is the partition must be self contained. A partition which accesses native methods on the device such as I/O devices would not be qualified as a legal partition and would not be migrated to a cloud server. The second requirement is that the partition must not access native states of the local machine. Partitions accessing native states below the application VM layer would not be legal. The third property prohibits nested partitions. Migration and re-integration points must be alternated. A thread which is migrated to the cloud server would be suspended on the client, and could not be suspended again until the thread is re-integrated from the cloud server. By applying the mentioned rules, a set of legal partition points can be obtained statically.

2.3.2. Migration and Re-integration

After partition points have been defined, the next step is to migrate the partitions to a remote machine for computation. Migration involves the recreation of the necessary runtime environment to execute the partition on the remote machine. For example, with VM level partitions, we could suspend the VM on the original machine and resume the snapshot on a remote machine [22]. With Java applications, the Java VM could be modified to ship objects to remote machines as proxy objects. The execution thread is created on the remote machine [23]. When remote computation is finished, the thread is re-integrated to the original machine.

2.3.3. Migration Trigger

We now have partition principles and migration-and-re-integration strategies. The remaining is to define a policy to where a given partition should be executed - remotely or locally. The condition for partitions to be migrated is when the benefits of executing the thread remotely outweighs the cost of partitioning, migration and transmission delays. The considerations could be computation time, battery consumption and network status. CloneCloud obtains these metrics by a dynamic profiler.
2.3.4. Comparing JSCloud with Existing Efforts

Existing efforts like CloneCloud offers augmented computation to specific platforms. In theory, web browsers (which is an application itself) could benefit from existing solutions as long as the platform is supported. When web applications could already benefit from existing solutions, how is JSCloud justified?

The matrices in Figure 2.6 shows the applicability of augmented computation solutions for different web applications (rows) on different platforms (columns). Existing solutions offers solutions per-platform. As an example, CloneCloud requires manual annotations to API methods for defining partition policies for each given platform [13]. Effort is needed to provide support for each platform. For example, different Android phone vendors ship their customized version of the Android OS and each version would require manual annotation.

![Matrices demonstrating the dimension of augmented computation support offered by per-platform and per-application solutions.](image)

Rather than requiring support to be rolled out to all target platforms, JSCloud aims provide a per-application solution that supports augmented computation across all platforms. This is due to the fact that JSCloud is actually implemented as standard-compliant JavaScript code which is supported by all JavaScript standard compliant browsers. This allows developers to design web applications with the assurance that augmented computation is available to all users.
3. System Design

The design of the proposed system JSCloud would be detailed in this chapter. JSCloud is a module between the web client and web server that provides improved JavaScript performance by means of code partitioning and migration.

3.1. System Overview

In an JavaScript web application, the program logic is delivered to the client as a JavaScript file from the web server. In order to support execution migration, the JavaScript file has to be modified to include the migration logic. This is performed by a module which automatically inserts such logic into the original code. The modified JavaScript file would be executed on the web browser, and when the migration logic is invoked, the code would be migrated to an JavaScript engine on the cloud for execution. Figure 3.1 describes the relation between the three modules: the JavaScript Code Modifier, the Modified JavaScript Code and the Cloud JavaScript Engine.

![Figure 3.1 System Overview of JSCloud](image)

Figure 3.1 System Overview of JSCloud
The **JavaScript Code Modifier** is an HTTP proxy which relays HTTP requests from and to the web server. This would allow JSCloud to be offered as a transparent service. When a JavaScript file is requested, the Cloud JavaScript Engine performs static analysis on the code to define legal partitions. The migration logic would be appended to the original JavaScript file, producing a **Modified JavaScript File** that is downloaded by the web browser.

The **Modified JavaScript File** is executed by the JavaScript engine of the web browser. The code contains logic which migrates the execution of partitions to the Cloud JavaScript Engine when the execution is estimated to be faster on the cloud. Such logic includes algorithms to estimate execution cost and stub functions which abstracts the migration of code.

The **Cloud JavaScript Engine** receives and executes migrated code. It consists of an instance of the Google V8 engine [24], which is one of the fastest open source JavaScript engines. The result of execution would be returned to the migration code, where the execution is reintegrated to the client JavaScript engine.

The work breakdown structure in Figure 3.2 fills the details of the three modules of JSCloud. The implementation of the modules would be documented in Section 4.

![Figure 3.2 Work breakdown structure of JSCloud](image-url)
3.2. Use Case Modeling

The use cases which JSCloud provide would be discussed in this section. The use case diagram in Figure 3.3 models the functionality provided by the system and shows their relation with the concerned actors.

Request file is the action performed by the web browser to request a file from the web server. As previously discussed, the JavaScript Code Modifier of the system is a HTTP proxy. When non-JavaScript files are requested, the file is simply relayed from the web server to the web browser. However, when a JavaScript file is requested, the JavaScript Code Modifier would append the migration logic into the file and transmit the modified code to the web browser.

The web browser would execute the modified instance of the JavaScript file. When the cost estimation logic in the modified code deems that a benefit is gained when code is executed remotely on the cloud, it performs the migration of execution.
### Use Case Description

**UC1 - Request File**
- relays non-JavaScript file HTTP requests between web browser and web server
- for JavaScript file requests, JSCloud provides web browser with a modified JavaScript code that contains the logic for execution migration

**UC2 - Migrate Execution**
- migrates execution of partitions from web browser to JSCloud

**Table 3.2 Description of use cases**

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC1 - Request File</td>
<td>- relays non-JavaScript file HTTP requests between web browser and web server&lt;br&gt;- for JavaScript file requests, JSCloud provides web browser with a modified JavaScript code that contains the logic for execution migration</td>
</tr>
<tr>
<td>UC2 - Migrate Execution</td>
<td>- migrates execution of partitions from web browser to JSCloud</td>
</tr>
</tbody>
</table>

Table 3.2 gives an overview of the two use cases the system provide. UC1 describes the proxy of the system which relays requests and modifies JavaScript files where appropriate. UC2 describes the migration of execution.

Table 3.3 and table 3.4 on the next page list the details of the two use cases, including the primary actors involved, the stakeholders and interests, the preconditions, the success guarantees and the scenarios.
UC1 - Request File

Primary actor - web browser

Stakeholders and interests - web browser want to obtain required resource from web server
- web server want to provide requestor with resource
- JSCloud want to provide web browser with modified code

Preconditions nil

Postcondition - web browser obtains required resource from web server

Main success scenario

<table>
<thead>
<tr>
<th>Actor action</th>
<th>System response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. web browser sends requests</td>
<td></td>
</tr>
<tr>
<td>2. JSCloud receives request</td>
<td></td>
</tr>
<tr>
<td>3. JSCloud sends request to web server</td>
<td></td>
</tr>
<tr>
<td>4. web server receives request</td>
<td></td>
</tr>
<tr>
<td>5. web server responds request</td>
<td></td>
</tr>
<tr>
<td>6. JSCloud receives response</td>
<td></td>
</tr>
<tr>
<td>7. JSCloud checks data type of resource</td>
<td></td>
</tr>
<tr>
<td>8. JSCloud sends response to web browser</td>
<td></td>
</tr>
<tr>
<td>9. web browser receives required resource</td>
<td></td>
</tr>
</tbody>
</table>

Extension - Request JavaScript File

7a. resource is a JavaScript file
7a.1. JSCloud defines legal partitions
7a.2. JSCloud appends cost estimation and migration code to JavaScript file
7a.3. continue at step 8

Table 3.3 Description of UC1 Request file
## UC2 - Migrate Execution

<table>
<thead>
<tr>
<th>Primary actor</th>
<th>- web browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholder and interest</td>
<td>- web browser want to migrate execution of a partition for better execution time</td>
</tr>
<tr>
<td>Precondition</td>
<td>- web browser is running the modified JavaScript code</td>
</tr>
<tr>
<td>Postcondition</td>
<td>- the partition is correctly executed remotely and reintegrated</td>
</tr>
<tr>
<td>Non-functional requirement</td>
<td>- the execution time is optimized</td>
</tr>
</tbody>
</table>

### Main success scenario

<table>
<thead>
<tr>
<th>Actor action</th>
<th>System response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Web browser executes JavaScript code</td>
<td></td>
</tr>
<tr>
<td>2. local engine estimates the cost for execution</td>
<td></td>
</tr>
<tr>
<td>3. local engine serializes the variables</td>
<td></td>
</tr>
<tr>
<td>4. local engine migrates the variables and partition to the cloud engine</td>
<td></td>
</tr>
<tr>
<td>5. cloud engine deserializes the variables and partitions</td>
<td></td>
</tr>
<tr>
<td>6. cloud engine executes the partition</td>
<td></td>
</tr>
<tr>
<td>7. cloud engine serializes result</td>
<td></td>
</tr>
<tr>
<td>8. cloud engine returns result to web browser</td>
<td></td>
</tr>
<tr>
<td>9. local engine receives result</td>
<td></td>
</tr>
<tr>
<td>10. local engine deserialize result and reintegrate execution</td>
<td></td>
</tr>
</tbody>
</table>

### Extension - Execution not migrated

- 2a. cost for migrated execution is higher than local execution
- 2a.1 execute locally

| Table 3.4 Description of UC2 Migration Execution |
3.3. Object Modeling

In this section, we would perform an analysis of the object oriented model of the system. Figure 3.4 is the class diagram of the system. It corresponds to the components as listed in the work breakdown structure in Figure in 3.2.

![Class diagram of JScloud](image)

Figure 3.4 Class diagram of JScloud

There are two relations that should be noted. CodeModifier is the creator of the ModifiedJavaScriptFile, which is downloaded by the web browser. The ExecutionMigrator and MigrationEndpoint reside in different systems, and therefore they communicate via message exchanges. Table 3.5 on the next page lists the descriptions of the classes.
Class | Purpose
---|---
JavaScriptCodeModifier | prepares Modified JavaScript Code
HTTPProxy | relays HTTP requests between client and web server
CodeModifier | appends cost estimation and migration logic to JavaScript code
CodeAnalyzer | defines legal partitions in original JavaScript code
ModifiedJavaScriptFile | contains web application code
CostEstimator | estimates the cost for migrated and local execution
ExecutionMigrator | migrates execution to cloud engine
CloudJavaScriptEngine | performs migrated execution
CodeExecutor | executes migrated code
MigrationEndpoint | accepts and replies migrated execution request

Table 3.4 Description of classes

3.4. High-Level Object Interaction Modeling

The high level object interaction of the classes mentioned in the previous section would be documented in this section.

Figure 3.5 Communication diagram of extension of UC1 - Request JavaScript File
The communication diagram in Figure 3.5 details the extension of Request JavaScript File of UC1. In this use case, when the web browser requests the resource from the web server, it would send the request to the HTTPProxy. The proxy then requests the resource from the web server and then subsequently receives it. Upon receiving the JavaScript file, the CodeAnalyzer would analyze the JavaScript file to define legal partitions and then the CodeModifier would append cost estimation and code migration logic to it. This Modified JavaScript File would then be returned to the web browser.

![Figure 3.6 Communication diagram of UC2 - Migrate Execution](image)

The communication diagram in Figure 3.6 shows the object interaction for UC2 Migrate Execution. When the web browser invokes a predefined partition, it would first invoke the CostEstimator, which estimates the time required to execute the partition locally and remotely. If the CostEstimator decides that migrated execution is justified, it would initiate the migration with the ExecutionMigrator. The ExecutionMigrator would transmit the serialized parameters and the partition to the MigrationEndpoint via HTTP message exchange. The MigrationEndpoint would invoke the CodeExecutor for the execution, and then it would reintegrate the results to the ExecutionMigrator. The execution of the partition is then completed.
4. Methodology and Implementation

In this chapter, the methodology and implementation would be documented. This chapter is divided into three sections, one for each of the three modules: the JavaScript Code Modifier; the Modified JavaScript Code; and the Cloud JavaScript Engine.

4.1. JavaScript Code Modifier

The JavaScript Code Modifier is an HTTP proxy with added functionalities of analyzing and modifying JavaScript files that it relays. It provides the web browser with a set of code that would migrate execution to the Cloud JavaScript Engine adaptively. The components of the JavaScript Code Modifier corresponds to the classes in Figure 3.5.

The JavaScript Code Modifier would be implemented in JavaScript and hosted on an node.js [25] instance. Node.js is a platform built on the Google V8 JavaScript engine with libraries for socket and I/O operations. It is highly suitable for running the JavaScript Code Modifier and the Cloud JavaScript Engine on. In this case of implementing an HTTP proxy, node.js could stream the bytes between web clients and web servers instead of treating each request as atomic events. As such, files could be streamed in real time. This allows a lower latency in providing the web browser with the resources. Node.js is further justified to be the platform of choice in the later section explaining why it is suitable for running the Cloud JavaScript Engine on.

4.1.1. HTTP Proxy

JSCloud provides its service with the JavaScript Code Modifier which is an HTTP proxy. The proxy could be located anywhere between the web browser and the web server. As such, JSRepeat could be located within the local area network (LAN), in the internet or in the server network as illustrated in Figure 4.1. Each configuration has its associated benefits and drawbacks which is tabulated in Table 4.1.
Figure 4.1 Locations where JSCloud could be located

<table>
<thead>
<tr>
<th>JSCloud Location</th>
<th>Benefits</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local area network</td>
<td>- minimal latency</td>
<td>- not applicable to mobile networks</td>
</tr>
<tr>
<td></td>
<td>- no proxy configuration required</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- JSCloud available to all users in the LAN for all web applications</td>
<td></td>
</tr>
<tr>
<td>Internet</td>
<td>- JSCloud available to all users for all web applications</td>
<td>- proxy configuration required</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- unoptimized latency</td>
</tr>
<tr>
<td>Server network</td>
<td>- no proxy configuration required</td>
<td>- unoptimized latency</td>
</tr>
<tr>
<td></td>
<td>- JSCloud available to all users of the web application</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Benefits and drawbacks for JSCloud configurations in general cases

In the LAN setting, JSCloud is available to all users. Latency is minimized as JSCloud is located in a local network as a forward proxy. JSCloud could also be offered as a public service on the internet, in which users could use it by configuring their proxy settings. The JSCloud proxy could also be a reverse proxy residing within the server network, making JSCloud available to all users of the web application.
4.1.2. Code Analyzer

The Code Analyzer is responsible for defining legal partitions. Partitions are segments of code that is between the entry and exit of a JavaScript function. There are two alternate schemes for code partition. In the first scheme, all JavaScript functions are defined as partitions. If we apply this scheme to the example in Figure 1.1, we would obtain three partitions: `merge_sort`, `split_array` and `merge`. In the second scheme, only JavaScript functions that are annotated by the web application developer are defined as partitions. Functions are annotated by adding a comment `//JSCLOUD MIGRATE` before the function declaration. Both schemes were implemented for testing but the first scheme faces performance issues (discussed in Section 5.1.2) and was excluded in the final version.

In the original design, partitions are defined according to the first scheme. The Code Analyzer would then determine whether each partition is legal. The legality refers to the eligibility of whether the partition could be migrated. Only legal partitions may potentially be migrated. JSCloud defines legal partitions as self contained partitions that each does not require external resources (e.g. global variables) and is not environment dependent (e.g. the `window` object). This is performed by static analysis. However as the first scheme was abandoned due to performance issues, the logic for checking the legality of partitions is demised. Under the second scheme, the programmer would manually annotate the JavaScript functions where he should make sure the function is self contained. Legality check is omitted to avoid the overhead. Although the legality of each partition is not guaranteed, the Modified JavaScript Code and Cloud JavaScript Engine would have two safeguards to ensure that any non-self contained partitions are appropriately executed. The two safeguards would be discussed in Sections 4.2.2 and 4.3.2.

Static analysis is used to determine the entry point and exit point of the JavaScript functions. The Original JavaScript File is searched for all occurrences of the `//JSCLOUD MIGRATE` annotation and the indexes would be stored into an array. The brackets `{ }` would then be analyzed to determine the entry point and exit point of the function.
Brackets `{` and `}` can exist in JavaScript code as part of comments or strings. These brackets are not part of the syntax and should be ignored. An alternate version of the JavaScript code would be constructed and stored in the string `alternateJSFile` where all comments and strings are replaced by white spaces of equal length. Comments are identified by characters enclosed between `//` and `\n` or `/*` and `*/`. Strings are identified by characters enclosed by `"` or `'`. In this alternate version of the code, all brackets are part of the syntax. The entry point and exit point of each annotated function could then be located by traversing the `alternateJSFile` string. Figure 4.2 shows the `merge_sort` algorithm with annotation. For illustration purposes the entry point and exit points are denoted by `^` and `$` respectively. The indexes of the entry points and exit points are stored into two arrays and passed to the Code Modifier for insertion of migration code.

```javascript
// JSCLOUD MIGRATE
function merge_sort(arr){
  function split_array(arr){
    ^ ENTRY POINT
    if (arr.length <= 1)
      return arr;

    var middle = parseInt(arr.length / 2);
    var left = arr.slice(0, middle);
    var right = arr.slice(middle, arr.length);

    return merge(split_array(left), split_array(right));
  }

  function merge(left, right){
    var result = [];
    while (left.length > 0 || right.length > 0){
      if (left.length > 0 && right.length > 0){
        if (left[0] <= right[0]){ // ENTRY POINT
          result.push(left.shift());
        } else {
          result.push(right.shift());
        }
      } else if (left.length > 0){
        result.push(left.shift());
      } else if (right.length > 0){
        result.push(right.shift());
      }
    }
    return result;
  }

  $ EXIT POINT
  return split_array(arr);
}
```

**Figure 4.2** Annotated version of the `merge_sort` code.
4.1.3. Code Modifier

The Code Modifier is responsible for inserting the code required for execution migration. The code is stored in three templates which are loaded by the Code Modifier and inserted into the appropriate locations of the Original JavaScript File to produce the Modified JavaScript File.

The `jsHeader.js` template contains the implementations of the Cost Estimator (Section 4.2.1), Execution Migrator (Section 4.2.2) and the JSON serialization library (Section 2.2). This template is prepended to the beginning of the Original JavaScript File.

The `jsStubHeader.js` template contains code that invokes the Cost Estimator and parameter type checker (Section 4.2.1). The decision of whether to migrate a partition or to execute locally is made here. This template is inserted in the entry point of each partition.

The `jsStubFooter.js` template contains code that records the execution times. The execution times are used by the Cost Estimator to calculate estimates (Section 4.2.1). This template is inserted in the exit point of each partition.

After inserting the code from the three template files into the Original JavaScript File, the Modified JavaScript File is produced and would be returned to the web browser by the HTTP Proxy. This completes of the extension of UC1 Generate Modified JavaScript File.

4.2. Modified JavaScript File

The Modified JavaScript File is a version of the original JavaScript file provided by the web server with migration logic added by the JavaScript Code Modifier. This section would document the implementation of the Cost Estimator and Execution Migrator. These two components are responsible for the initiation of migration.
4.2.1. Cost Estimator

The Cost Estimator is responsible for deciding whether to execute a partition locally or remotely. The decision is made based on execution time. Execution time grows at different rates in relationship to partition size on different platforms. This forms the basis of this project - to migrate the execution when a remote execution has a shorter execution time.

![Figure 4.3 Relationship between partition size and execution times](image)

A partition is migrated when the time for migration plus the time for remote execution is shorter than the time for local execution. The migration delay is the time required to transmit messages between the web browser and the Cloud JavaScript Engine. If there is no migration overhead, remote execution could be assumed to be always faster or as fast as local execution. In other words, the migration overhead delay is the only cost of migration.

Assume $d$ to be the migration overhead delay. $d$ is not a constant value during the whole session. This is especially true on mobile networks where a big fluctuation could be expected. The Cost Estimator would estimate $d$ based on recent migration overhead delays.
After estimating the migration overhead $d$, the Cost Estimator would decide if it is worthwhile to pay the cost. Assume that the time for remote execution to be $t$, then the time for local execution would be $t + k$, where $k$ is the difference in the execution time; and the total time for migrated execution would be $t + d$. We migrate the execution when the total time for migrated execution is less than the total time for local execution. That is we migrate when $t + d < t + k$, which is $d < k$ (The intersection of the two curves in figure 4.2 is $d = k$). In other words, it is worthwhile to migrate when $d$ is smaller than $k$.

The value of $k$ would be estimated by the Cost Estimator based on the execution times of recent local and remote executions. The Cost Estimator would maintain a record of $k$ in relation to the block size of the parameters (under the assumption that $k$ is in direct proportion to the block size of parameters). When a partition of parameter block size $n$ is migrated to the cloud engine for execution, the Cost Estimator would time the execution and the migration overhead. If we subtract the overhead time from the execution time, we obtain the remote execution $t$ for block size $n$. When a partition of block size $n$ is executed locally, the Cost Estimator would also time the local execution time, which is $t + k$ for parameter block size $n$. Having obtained sufficient values of $t$ in relation to $n$ and $t + k$ in relation to $n$. We could extrapolate the value of $t$ and $t + k$ for a given $n$, which gives the value of $k$ for a given $n$. The more data we have on local and remote execution times, the more accurate the estimation of $k$ would be.

The relation between $k$ and $n$ is maintained individually for each legal partition. This is essential as the effect of block size to execution time is different for each partition. As an example, consider two functions power($a,b$) and add($a,b$). The function power raises $a$ to the power of $b$ while the function add sums $a$ and $b$. Both functions take parameters of similar block size but the rate of increase in execution time is very different. Therefore the performance metrics of $k$ in relation to $n$ should be kept individually for each partition.
As $k$ is the execution time, it is independent of network transmission delays. For each invoked partition of parameter block size $n$, the Cost Estimator would estimate the value of $k$ and the migration overhead $d$. When $d < k$, the partition would be migrated to the cloud engine by the Execution Migrator.

When the web application is just loaded, the Cost Estimator would not have sufficient samples to estimate the local execution time $t + k$, remote execution time $t$ and migration overhead $d$. The flowchart in Figure 4.4 depicts the process which the Cost Estimator makes migration decisions.

![Flowchart depicting the decision making process of the Cost Estimator](image)

**Figure 4.4** *Flowchart depicting the decision making process of the Cost Estimator*

The Cost Estimator would first check if there is sufficient samples to estimate the local execution time. When the web application is just loaded, no partitions are executed and therefore we do not have any samples of local execution times $t + k$. The Cost Estimator would therefore make the decision to execute the partition locally. When there are more than two samples of $t + k$ of block
sizes $x$ and $y$ where $x \neq y$, the Cost Estimator would attempt to estimate the $t_n + k_n$ for the given block size $n$. The estimation $e_{local}$ for block size $n$ is performed by linear interpolation if $x \leq n \leq y$ or otherwise by linear extrapolation. For linear interpolation, the samples of which $x$ and $y$ are closest to $n$ is chosen for the interpolation. For linear extrapolation, the two samples $x$ and $y$ of smallest or largest block sizes are accordingly chosen for the extrapolation. If $e_{local}(n) < 0$, then the Cost Estimator would treat the situation as having insufficient samples for estimation and would decide to execute the partition locally. $e_{local}(n) < 0$ when the slope $m$ of the line connecting $(x, t_x + k_x)$ and $(y, t_y + k_y)$ is sufficiently big or when the slope $m$ is negative. Both cases signify that the estimation is inaccurate and therefore the Cost Estimator would decide to execute locally. This decision for local execution would also provide an additional set of local execution time $t_n + k_n$ to refine future estimations.

Using the same estimation algorithm as $e_{local}$, the Cost Estimator would attempt to estimate the remote execution time $t_n$ for given block size $n$. If there are insufficient samples for the estimation, the Cost Estimator would proceed to check if $e_{local}(n) > 100ms$. This threshold is set arbitrarily. The partition is migrated when $e_{local}(n) > 100ms$. Otherwise it is executed locally.

If the Cost Estimator has sufficient data to estimate $t_n$ then it follows that there are sufficient data to estimate $d_n$ as $t$ and $d$ are obtained together in the measurement of migrated execution times (i.e. number of samples for $t$ = number of samples for $d$). The Cost Estimator would then proceed to estimate $t_n$ and $d_n$. $k_n$ is then calculated ($e_{local}(n) - e_{remote}(n) = t_n + k_n - t_n = k_n$). If $d_n < k_n$ then the partition is migrated. Otherwise it is executed locally.

The estimation of the overhead $d_n$ for block size $n$ is based on 100 recent overhead timing samples (or as much as available if there is less than 100 samples). Assume that $s$ samples are available where $s < 100$. We first calculate the weighted average $r$ of $d_x / x$ (i.e. overhead time per block size) where $x = 1, 2, ... s$. The weight for $d_x / x$ is calculated as $timestamp_x - timestamp_{x-1}$ where $timestamp_x$ is the time when the sample for $d_x$ is obtained (i.e. more recent samples have higher weight). We then multiply $r$ by $n$ to obtain $d_n$.  

30
4.2.2. Execution Migrator

The Execution Migrator is responsible for the migration of partitions to the cloud engine. The code for Execution Migrator is inserted into the Modified JavaScript File by the Code Modifier (Section 4.1.3). The Execution Migrator provides two methods: `migrate` that migrates the partition to the Cloud JavaScript Engine; and `checkArgumentsType` that checks the data type of the arguments (this is the first safeguard against the migration of non-self-contained partitions as mentioned in 4.1.2).

The pseudocode for the `migrate` method is shown in Figure 4.5. The `migrate` method accepts the `args` parameter which is a JavaScript arguments object. JavaScript parameters passed into any function is contained in the `arguments` object and can be accessed within the scope of the function body. The `migrate` method would first serialize the arguments and the function which is accessed by `args.callee` into JavaScript Object Notation (JSON) with the external library mentioned in Section 2.2. An XML-HTTP request would be sent to the cloud engine url for the migration of arguments `a` and method `m`. The `async=false` statement denotes that the call is asynchronous. This is performed as the thread should be blocked while awaiting the result from the cloud engine. This, however, should not be a concern as the migrated execution should be quicker than executing the partition locally, which would also hold the thread anyhow. If an asynchronous call is to be used, the whole web application code would have to be redesigned with substantial changes to cater for the callbacks, which is out of the hands of the current version of our JavaScript Code Modifier.

```javascript
ExecutionMigrator.migrate = function(args){
    var a = serialize(args)
    var m = serialize(args.callee)
    var result = ajaxCall('async=false', endpointURL, a, m);
    return deserialize(result);
}
```

**Figure 4.5** Pseudocode of the `migrate` method
The pseudocode for `checkArgumentsType` is shown in Figure 4.6. The method accepts the `arguments` object `args` as the parameter and returns a boolean value. The `args` object is first casted into an array `a`. Then the elements in `a` are iteratively checked. If an element in `a` is found to be a non-number, non-boolean, non-String, non-array or is undefined, the method returns false and the partition would not be migrated. `checkArgumentsType` returns true when all parameters used by the partition are primitive data types and the partition is eligible for migration.

```javascript
ExecutionMigrator.checkArgumentsType = function(args){
    var a = args.toArray()
    for (i in a){
        if (typeof(a[i]) != 'number' && typeof(a[i]) != 'string' && typeof(a[i]) != 'boolean' && typeof(a[i]) != 'undefined' && !(a[i] instanceof Array)){
            return false;
        }
    }
    return true;
}
```

**Figure 4.6 Pseudocode of the checkArgumentsType method**

### 4.3. Cloud JavaScript Engine

The Cloud JavaScript Engine (refer to Figure 3.2) is the module which accepts execution migration requests from the Modified JavaScript Engine. It is a stateless system which processes the requests individually. The Cloud JavaScript Engine is implemented in JavaScript on the node.js platform. (Node.js was briefly introduced in Section 4.1). Node.js is suitable for a couple of reasons. Firstly, as node.js is written in JavaScript, it would readily parse the migration requests which are serialized in the JSON format. Secondly, Node.js is designed to handle a large number of simultaneous connections while providing a high requests/second throughput [26]. Thirdly, the migrated executions could be executed directly on node.js, taking advantage of its V8 JavaScript engine which has high performance as demonstrated in the benchmarks in table 1.1. The Cloud JavaScript engines contains two parts: the Migration Endpoint and the Code Executor.
4.3.1. Migration Endpoint

The Migration Endpoint is the listener for requests. Upon receiving migration requests from the web browser, it would deserialize the parameters and submit the execution to the Code Executor. When the Code Executor finishes execution, it would serialize the result and return it to the web browser.

4.3.2. Code Executor

The Code Executor accepts migrated partitions and executes them. As the required parameters and method are already prepared by the Migration Endpoint, the Code Executor would simply invoke the method with the parameters and return the result to the Migration Endpoint. If the execution of the migrated partition causes a runtime error, it would throw an `MigratedExecutionFailed` exception to the Execution Migrator of the Modified JavaScript Code. The Execution Migrator would then prohibit the partition from being migrated again and resume to re-execute the partition locally. This is the second safeguard mentioned in Section 4.1.2.

UC2 Migrate Execution is completed when the Migration Endpoint returns the result to the Execution Migrator and the execution is reintegrated to the local runtime.
5. Performance Evaluation

The performance of JSCloud would be evaluated in this section. This section covers three aspects of performance: the overhead introduced by JSCloud; the accuracy of the Cost Estimator; and the overall performance gained by JSCloud.

5.1. Overhead Evaluation

There are two types of overheads in JSCloud. The preparation overhead is the overhead for preparing the Modified JavaScript File by the JavaScript Code Modifier (UC1). The execution overhead is the overhead for executing partitions in the Modified JavaScript File (UC2).

5.1.1. Preparation Overhead

The HTTP Proxy of JSCloud is the first source of preparation overhead. Requested files are relayed through the proxy and a latency would be introduced. Moreover, additional overhead is incurred when a JavaScript file is requested as the JavaScript file would be downloaded and uploaded instead of streamed through the proxy. This overhead is highly variable as it is dependent on the network condition of the user and the location of where JSCloud is located (as depicted in Figure 4.1).

The second source of preparation overhead is the time used for static analysis and code modification in the Code Modifier. The average processing time for each partition to be defined and included with migration logic is 0.35ms. The number is derived from a series of samples as shown in Table 5.1.

<table>
<thead>
<tr>
<th>number of partitions</th>
<th>130</th>
<th>50</th>
<th>12</th>
<th>8</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>processing time (ms)</td>
<td>64</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 5.1 Samples of time spent processing a JavaScript file of a number of partitions
Furthermore, the Modified JavaScript File would have a larger file size than the Original JavaScript File due to the additional code (Section 4.1.3). This larger file size is the third source of preparation overhead as the downloading time would be increased. The size of the additional code is tabulated in Table 5.2.

<table>
<thead>
<tr>
<th>code size (bytes)</th>
<th>minified code size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>file header</td>
<td>35150</td>
</tr>
<tr>
<td>stub footer</td>
<td>3913</td>
</tr>
<tr>
<td>stub footer</td>
<td>660</td>
</tr>
</tbody>
</table>

Table 5.2 Additional code size of each part of the Modified JavaScript File

The file header is added to each Modified JavaScript File. The stub header and stub footer is added to each partition of the file. The code size could be reduced by minification, which is the process of removing unnecessary white spaces, newlines and comments in the JavaScript code.

To analyze the overall overhead in the preparation of the Modified JavaScript File, an experiment was performed to obtain the actual overhead measurements when JSCloud is used as a reverse proxy (refer to Figure 4.1). The time required to obtain JavaScript files of 1, 10, and 100 partitions from a web server with and without JSCloud were recorded. By computing the difference between the time used for obtaining the file with JSCloud and the time without JSCloud, the overhead is obtained. Table 5.3 shows the overhead data.

<table>
<thead>
<tr>
<th>number of partitions</th>
<th>without JSCloud (ms)</th>
<th>with JSCloud (ms)</th>
<th>overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>73</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>138</td>
<td>88</td>
</tr>
<tr>
<td>100</td>
<td>309</td>
<td>805</td>
<td>496</td>
</tr>
</tbody>
</table>

Table 5.3 Time required to complete request for JS file of different number of partitions

The time required to complete the file request is approximately doubled when JSCloud is used. While this overhead is not insignificant, in actual usage the delay is acceptable in terms of webpage load times. To put the figures into perspective, the homepage of Google took 1150ms to load (including all images and scripts) on the same experiment setup.
5.1.2. Execution Overhead

This section would analyze the two types of execution overheads: the execution overhead in executing a partition locally; and the migration overhead when a partition is executed remotely.

In each partition of the Modified JavaScript File are the additional logic inserted by the Code Modifier (Section 4.1.3). These logic are used to determine whether the execution of a partition should be migrated and to record timing data. It is executed regardless of whether the partition was migrated and is the only source of execution overhead in the local execution of a partition.

To analyze the overhead of executing a partition locally, the execution times for sorting an array of different sizes using the `merge_sort` algorithm in Section 1.2 were measured in two runs. The first run is executed with the Original JavaScript File to obtain the baseline measurements. The second run is executed with a special version of Modified JavaScript File which forces all executions to not migrate, thus obtaining the local execution time + the overhead. By subtracting the time of the second run by the first run, the overhead could be obtained. The results are shown in Table 5.4.

<table>
<thead>
<tr>
<th>array size</th>
<th>original JS code (ms)</th>
<th>modified JS code (ms)</th>
<th>overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>49</td>
<td>48</td>
</tr>
<tr>
<td>1,000</td>
<td>24</td>
<td>1975</td>
<td>1951</td>
</tr>
<tr>
<td>10,000</td>
<td>1627</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.4 Time required to sort arrays of different sizes with Original and Modified JS codes

The results show that the overhead in the execution of partitions in the Modified JavaScript File is very high. The sorting of 10,000 elements in the Modified JavaScript File resulted in a browser timeout. The overhead caused the local executions of partitions to be too slow.
The reason for the significant overhead was investigated. As we have discussed in Section 4.1.2, there are two schemes for the Code Analyzer. The first scheme would automatically define all methods as legal partitions. The second scheme would define partitions as per the annotations in the JavaScript file. The figures in this experiment were obtained using the first design. All methods in the Modified JavaScript File were defined as legal partitions. Consequently, when the merge sort method was invoked recursively, the Cost Estimator was also invoked in each recursion for a large number of times. This caused the long execution times in the Modified JavaScript Code. The first scheme is therefore abandoned.

The same experiment was performed with the second design of the Code Analyzer where only annotated methods would be defined as legal partitions. Only the method that invokes the `merge_sort` method was annotated for migration (as in Figure 4.2). As with the previous experiment, the Modified JavaScript File is a special version where execution is forced to be performed locally. The results are tabulated in Table 5.5.

<table>
<thead>
<tr>
<th>array size</th>
<th>original JS code (ms)</th>
<th>modified JS code (ms)</th>
<th>overhead (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1,000</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>1627</td>
<td>1632</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.5 Time required to sort arrays of different sizes with Original and Modified JS codes where Code Analyzer defines only annotated methods as legal partitions.

The figures show that the overhead is negligible when the Code Analyzer defines only annotated methods as partitions. This is due to the fact that the Cost Estimator was invoked for one time only as the annotated method was executed once only.

The two experiments shows that in the local execution of a partition, the automated partitioning of all methods by the Code Analyzer would result in the local execution time being much longer than the Original JavaScript Code. JSCloud is only practical when the Code Analyzer partitions as per the annotations in the JavaScript file.
The migration overhead would then be discussed. This overhead includes the serialization of the request; the deserialization of the request; the serialization of the reply; the deserialization of the reply; and network transmission. In other words, the overhead is $t_1 + t_2 + t_3 + t_5 + t_6 + t_7$ in Figure 5.1.

![Figure 5.1 Lifecycle of a migrated execution](image)

Experiments were carried out to measure the overhead. The partition entry and exit times were recorded to calculate the time used to execute the partition (i.e. $t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7$). The time taken for the remote execution is also measured (i.e. $t_4$). By subtracting the two measurements, we obtain the overhead $t_1 + t_2 + t_3 + t_5 + t_6 + t_7$. Tables 5.6 and 5.7 list the migration overhead measured from a notebook computer and mobile phone respectively. JSCloud was configured as a reverse proxy.

<table>
<thead>
<tr>
<th>array size</th>
<th>1994</th>
<th>2683</th>
<th>3543</th>
<th>4012</th>
<th>5261</th>
<th>6674</th>
<th>7590</th>
<th>8105</th>
</tr>
</thead>
<tbody>
<tr>
<td>migration overhead (ms)</td>
<td>215</td>
<td>141</td>
<td>934</td>
<td>261</td>
<td>218</td>
<td>377</td>
<td>251</td>
<td>275</td>
</tr>
</tbody>
</table>

**Table 5.6** Migration overhead measured from a MacBook Pro for sorting arrays of different sizes by accessing JSCloud over the internet on a cable modem connection

<table>
<thead>
<tr>
<th>array size</th>
<th>1213</th>
<th>2302</th>
<th>2575</th>
<th>3095</th>
<th>3365</th>
<th>3978</th>
<th>4615</th>
<th>4904</th>
</tr>
</thead>
<tbody>
<tr>
<td>migration overhead (ms)</td>
<td>838</td>
<td>1219</td>
<td>1295</td>
<td>1541</td>
<td>1431</td>
<td>1741</td>
<td>1898</td>
<td>1932</td>
</tr>
</tbody>
</table>

**Table 5.7** Migration overhead measured from a BlackBerry 9780 for sorting arrays of different sizes by accessing JSCloud over the internet on a 3G connection
The figures in Table 5.6 and 5.7 show that the migration overhead is generally in proportion to the array size. However, as transmission delay fluctuates, there are cases where a larger array had a shorter migration overhead, such as in Table 5.6 where the sorting of 3543 elements required an overhead of 934ms whereas the sorting of 4012 elements required an overhead of only 261ms.

The migration overhead is large when the array size is large and the network is not stable. As discussed in Section 4.2.1, the migration overhead is used to determine whether a partition should be migrated.

5.2. Accuracy of Cost Estimator

The Cost Estimator is responsible for making decisions on whether a partition should be migrated or not based on the history of execution times and migration overheads in the session. As such, the accuracy of Cost Estimator determines the efficiency of JSCloud. Two aspects would be analyzed in this section: the accuracy of the estimations; and the accuracy of the decisions.

5.2.1. Accuracy of Estimations

The Cost Estimator would calculate the estimated local execution time (inclusive of the local execution overhead discussed in Section 5.1.2), estimated remote execution time ($t_4$ in Figure 5.1) and the estimated migration overhead ($t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7$ in Figure 5.1). The estimation is based on historical data, and therefore the estimation should be progressively more accurate as more samples are recorded.

To measure the accuracy, 50 arrays of random length from 1 to 5000 containing random numbers were sorted with the merge sort algorithm in Section 1.2 and the execution times were recorded.
The experiment was performed on a BlackBerry 9780 connected to the Internet on a 3G connection. JSCloud is configured as a reverse proxy (refer to Section 4.1.1). Three trial runs were taken and the data is tabulated in Tables 5.9, 5.10 and 5.11 (starting on the next page). A control sample was obtained by running the experiment with the Original JavaScript File to obtain the baseline local execution times. This control data is recorded in the baseline column.

The blank cells in the estimated data columns denotes that an estimation was not possible due to insufficient samples (see Section 4.3.1). A partition is either executed locally or migrated, therefore in each row only the actual local time or the actual migrated time + actual overhead could be recorded. The shaded rows are excessive migrations and would be discussed in Section 5.2.2.

Table 5.8 tabulates the mean, median and standard deviation of the percentage error of the estimations in the three trials.

<table>
<thead>
<tr>
<th></th>
<th>local time estimation percentage error</th>
<th>remote time estimation percentage error</th>
<th>overhead estimation percentage error</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>7.73%</td>
<td>4.99%</td>
<td>3.51%</td>
</tr>
<tr>
<td>median</td>
<td>2.01%</td>
<td>5.40%</td>
<td>2.30%</td>
</tr>
<tr>
<td>standard deviation</td>
<td>25.09%</td>
<td>32.44%</td>
<td>27.57%</td>
</tr>
</tbody>
</table>

*Table 5.8 Mean, median and standard deviations of the percentage errors of the estimations*

As the Cost Estimator makes the decision of whether to migrate based on the estimated data, inaccurate estimations would affect the accuracy of the decision. Section 5.2.2 would look into accuracy of the migration decision.
Table 5.9 Actual and estimated execution times and overheads recorded when sorting a 50 number array on a BlackBerry 9780 on a 3G connection to the internet. First trial.
Table 5.10 Actual and estimated execution times and overheads recorded when sorting a 50 number array on a BlackBerry 9780 on a 3G connection to the internet. Second trial.
Table 5.11  Actual and estimated execution times and overheads recorded when sorting a 50 number array on a BlackBerry 9780 on a 3G connection to the internet. Third trial.
5.2.2. Accuracy of Decisions

In Section 5.2.1 we have discussed the accuracy of estimations. Based on these estimations the Cost Estimator would make a decision on whether the execution of a partition should be migrated. The accuracy of the decision would be discussed in this section.

As discussed in Section 4.2.1, the Cost Estimator would decide to migrate the execution of a partition when \( d < k \) (i.e. estimated local execution time - estimated remote execution time < estimated overhead). However, as both the value of \( d \) and \( k \) are based on estimation, the Cost Estimator could possibly arrive at a wrong decision when the values of \( d \) and \( k \) are inaccurate.

When a partition is executed locally, we could not judge on the correctness of the decision as we do not have the actual timing for the migrated execution of the partition. In other words, we cannot claim in hindsight that we should have migrated for a shorter execution. As an example, at time \( t = 0 \), the Cost Estimator decides to execute partition \( x \) locally. The execution completes at \( t = 5 \). To conclude that the decision is wrong, we have to show that if the execution was migrated at \( t = 0 \), it would complete before \( t = 5 \). However, the migration overhead at \( t = 0 \) would never be known as it is dependent on the network condition at that instant. Thus, we could not determine in hindsight that a decision to execute locally is wrong.

For migrated partitions, we could determine the correctness by comparing the migrated execution time with a baseline local execution time. The baseline time is recorded in the execution of a special version of the Modified JavaScript File which forces all executions to be performed locally. Local execution times do not suffer from the fluctuations of network conditions and would serve as a reliable comparison.

We could conclude that a partition is correctly migrated when execution time is shortened. On the contrary, a partition is excessively migrated when the migrated execution takes longer than executing it locally. The shaded rows in Tables 5.9, 5.10 and 5.11 are excess migrations where
the baseline local execution time < actual remote execution time + actual overhead time. Table 5.12 tabulates the number of excessive migrations in Tables 5.9, 5.10 and 5.11.

<table>
<thead>
<tr>
<th></th>
<th>trial 1</th>
<th>trial 2</th>
<th>trial 3</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of excessive migrations</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>number of migrations</td>
<td>26</td>
<td>11</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>percentage of excessive migrations</td>
<td>23.08%</td>
<td>36.34%</td>
<td>35.71%</td>
<td>31.38%</td>
</tr>
</tbody>
</table>

Table 5.12 Statistics on excessive migrations

Statistics show that 31.38% of the migrations in the three trial runs are excessive. Excessive migrations could be occur because of inaccurate estimations or a surge in overhead due to unstable network conditions.

Items 4 and 5 of the three trials are excessively migrated because of insufficient timing samples. The estimated remote execution time and estimated overhead could not be calculated as there were insufficient samples of remote execution times. At least two points are required for the interpolation/extrapolation. The partition was migrated because the estimated local time exceeds the arbitrary threshold of 100ms (refer to Section 4.2.1).

The excessive migrations would cause the total execution time to be increased. The increased in execution time would be discussed in Section 5.3 where the performance gain delivered by JSCloud would be discussed.
5.3. Performance Gain

Section 5.2.2 discussed how excessive migrations would increase the total execution time. The performance gain delivered by JSCloud would be discussed in this section.

The performance gain is analyzed based on three measurements obtained from Tables 5.9, 5.10 and 5.11. First, the execution time without JSCloud which is the sum of the baseline local execution time. Second, the execution time with JSCloud which is the sum of actual local, remote and overhead times. Third, the ideal execution time which is the execution time when no excessive migrations were performed. This ideal execution time is calculated by taking the sum of the execution times for local executions; the execution times for correct migrated executions; and the baseline local execution time for excessive migrations. The three measurements are tabulated in Table 5.13.

<table>
<thead>
<tr>
<th></th>
<th>trial 1</th>
<th>trial 2</th>
<th>trial 3</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution time without JSCloud (ms)</td>
<td>77373</td>
<td>77373</td>
<td>77373</td>
<td>77373</td>
</tr>
<tr>
<td>execution time with JSCloud (ms)</td>
<td>53320</td>
<td>74554</td>
<td>70236</td>
<td>66036.67</td>
</tr>
<tr>
<td>ideal execution time with JSCloud (ms)</td>
<td>51936</td>
<td>66371</td>
<td>67102</td>
<td>61803</td>
</tr>
</tbody>
</table>

Table 5.13 Statistics on execution times

In Table 5.14, the same data is presented as the percentage relative to the execution time without JSCloud.

<table>
<thead>
<tr>
<th></th>
<th>trial 1</th>
<th>trial 2</th>
<th>trial 3</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution time without JSCloud (ms)</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>execution time with JSCloud (ms)</td>
<td>68.91%</td>
<td>96.36%</td>
<td>90.78%</td>
<td>85.35%</td>
</tr>
<tr>
<td>ideal execution time with JSCloud (ms)</td>
<td>67.12%</td>
<td>85.78%</td>
<td>86.73%</td>
<td>79.88%</td>
</tr>
</tbody>
</table>

Table 5.14 Statistics on execution times shown as relative percentages

The experiment results shows that the execution time was reduced by 14.65%. The excess migrations caused by incorrect decisions made by the Cost Estimator has prevented a potential 5.47% reduction in execution time.
The same experiment used to obtain the data in Tables 5.9, 5.10 and 5.11 was repeated on different devices to obtain the performance statistics of JSCloud in different scenarios. The results are presented in Figure 5.2.

Figure 5.2 Statistics on execution times shown as relative percentages on various devices. Lower number is better.
To account for the differences in execution performance of the devices, the length of the arrays were adjusted individually for each device such that the sorting takes around 77000ms without JSCloud. In other words, 100% equals to 77000ms of execution time for all devices.

The MacBook Pro and iPod Touch were connected to the Internet via Wi-Fi to a router that accesses the internet from a cable modem. The BlackBerry, iPhone 4S and Galaxy S2 (Android device) were connected to the internet via 3G network.

In the case of MacBook Pro and iPod Touch, JSCloud brought the execution time down to 57.95% and 52.15% of the original respectively. The execution was almost twice as fast. The performance gained by JSCloud on these 2 devices are high due to two reasons: the internet connection is fast and stable; and the JavaScript engine in the two devices are slow.

For the BlackBerry 9780, JSCloud brought the execution down to 85.35% of the original. The lower performance gain is due to the high latency and instability of the 3G network.

On the iPhone 4S and Galaxy S2, JSCloud had increased the execution times to 112.85% and 105.85% respectively. The JavaScript engines in these two devices are already efficient enough and any migration would suffer from the high latency of the 3G network. The ideal execution time being 100% suggests that the best performance is achieved when no partitions are migrated (i.e. when JSCloud is not used).
6. Conclusion

This section is a summary of the JSCloud project. The discussion would be divided into four sections: the achievements; the critical review; the limitations; and future developments.

6.1. Achievements

The performance evaluation (Section 5) of JSCloud demonstrated that part of the aims were successfully achieved.

JSCloud had successfully improved the execution time of JavaScript intensive web applications on devices that have a lower JavaScript performance. The execution time could be reduced by as much as 47.85% (Section 5.3).

The overhead for preparing the Modified JavaScript Code is negligible in comparison to the average loading time of a webpage (Section 5.1.1).

6.2. Critical Review

Although the performance evaluation (Section 5) showed positive results, the analysis also revealed two issues with JSCloud.

JSCloud was originally intended to provide performance benefits to all existing web applications as shown in Figure 2.6. However, when using the first scheme (Section 4.1.2) of partitioning which defines all functions in the Original JavaScript File as eligible for migration, the overhead for local executions were too high (Section 5.1.2). The second scheme where only annotated partitions was finally adopted. This scheme would require the manual annotation of partitions by the web developer.
JSCloud also increased the execution time on devices which already have a high processing power (Figure 5.2). This performance issue brings the practicality of JSCloud into doubt as the future mobile devices would become faster and faster. The high JavaScript performance of the current generation of smartphones (iPhone 4s and Samsung Galaxy S2) has already defeated the purpose of JSCloud. The benefits of JSCloud is only limited to devices and web browsers with low JavaScript performance.

6.3. Limitations

JSCloud is accessed as an HTTP Proxy. As such, JSCloud is incompatible with the HTTPS protocol as the data stream is encrypted. Furthermore, some web servers would compress the reply to the resource requests from web browsers to gzip format to minimize the transit time. The current version of JSCloud would state explicitly in the HTTP request header that gzip is not accepted such that the web server would reply only in plaintext. This incurs an overhead to the transmission time as compression is not used.

Partitions also must be self-contained (Section 4.1.2) for migration. Therefore the implementation of web applications must have a clear separation of control classes and boundary classes to benefit from JSCloud. Control classes should be responsible for processing only and must not access the DOM. Existing web applications where control classes would access the web browser environment would have to be refactored in order to meet the self-contained criteria.
6.4. Future Developments

The future developments of the JSCloud should emphasize on improving the issues in performance and practicability as mentioned in Section 6.2.

The issue of performance degradation on already JavaScript efficient devices would be addressed. When the HTTP Proxy receives a page request from an already powerful device it would not insert migration logic to the code. The detection is performed by inspecting the HTTP request header. The attribute \textit{User-Agent} would state the name of the device as well as the make and version of the web browser. If the HTTP Proxy receives a request from an \textit{User-Agent} that is in a predefined list of JavaScript efficient devices, then the JavaScript file is left unmodified.

In JSCloud, the bottleneck in the performance of migrated executions is the transmission time in the migration overhead (Section 5.1.2). The overhead could be further reduced by compressing the reply from the Cloud JavaScript Engine into gzip format by JSCloud.
References


52


Appendix

A. Monthly Log

<table>
<thead>
<tr>
<th>Month</th>
<th>Progress Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>October 2011</td>
<td>- literature review</td>
</tr>
<tr>
<td></td>
<td>- produce project plan</td>
</tr>
<tr>
<td>November 2011</td>
<td>- literature review</td>
</tr>
<tr>
<td></td>
<td>- feasibility test</td>
</tr>
<tr>
<td>December 2011</td>
<td>- literature review</td>
</tr>
<tr>
<td></td>
<td>- use case modeling</td>
</tr>
<tr>
<td>January 2012</td>
<td>- object modeling</td>
</tr>
<tr>
<td>February 2012</td>
<td>- object modeling</td>
</tr>
<tr>
<td></td>
<td>- algorithm design</td>
</tr>
<tr>
<td></td>
<td>- produce interim report</td>
</tr>
<tr>
<td></td>
<td>- implementation</td>
</tr>
<tr>
<td>March 2012</td>
<td>- implementation</td>
</tr>
<tr>
<td></td>
<td>- system testing</td>
</tr>
<tr>
<td></td>
<td>- measure performance data</td>
</tr>
<tr>
<td></td>
<td>- evaluate performance data</td>
</tr>
<tr>
<td>April 2012</td>
<td>- evaluate performance data</td>
</tr>
<tr>
<td></td>
<td>- produce final report</td>
</tr>
</tbody>
</table>

B. User Guide

Cloud.js executes on the node.js runtime which could be obtained from http://nodejs.org/. After the successful installation node.js, JSCloud could be started by the following procedures:

1. Configuring the HTTP Proxy

As mentioned in Section 4.1.1, there are multiple configurations for the HTTP Proxy. JSCloud provides the option to set the HTTP Proxy as a standard proxy or a reverse proxy. This setting is configured by modifying the `var forwardProxy = true` line in HTTPProxy.js.
The default port for the HTTP Proxy is 8000. To change the listening port of the HTTP Proxy, modify the `var listeningPort = 8000` line in HTTPProxy.js.

2. Configuring the Cloud JavaScript Engine

The default port for the Cloud JavaScript Engine is 8001. To change the listening port of the Cloud JavaScript Engine, modify the `var cloudEnginePort = 8001` line in HTTPProxy.js. Then modify the `var listeningPort = 8001` line in MigrationEndpoint.js.

3. Starting the HTTP Proxy

The HTTP proxy could be started by the command `node HTTPProxy.js` in the command line utility of the operating system.

4. Starting the Cloud JavaScript Engine

The Cloud JavaScript Engine could be started by the command `node MigrationEndpoint.js` in the command line utility of the operating system.

JSCloud should then be successfully started. If JSCloud is configured as a normal proxy, any devices that has added the JSCloud HTTP Proxy to the internet settings would benefit from JSCloud. If JSCloud is configured to be a reverse proxy, devices only have to access the web application normally by typing in the URL in the web browser.