Journal of Web Engineering, Vol. 11, No.4 (2012) 337-349
© Rinton Press

REUSE OF JIT COMPILED CODE BASED ON BINARY CODE PATCHING IN JAVASCRIPT ENGINE

SANGHOON JEON

KT innotz, Seoul

shjeun@ktinnotz.com

JAEYOUNG CHOI

Soongsil University, Seoul

choi@ssu.ac.kr

Received September 3, 2011
Revised August 7, 2012

JavaScript is a core language of web applications. As the most frequently used web language, it is used in more than 90% of web pages around the world. As a result, the performance of JavaScript engines becomes an important issue. In order to increase the execution speed of web applications, many JavaScript engines are embedded in JIT (Just-in-time) Compiler. However, JIT compilers are required to execute and compile applications at the same time. Therefore, this technique has been hardly applicable to embedded systems, in which system resources are limited. In this paper, we present a reusing technique of JIT compiled code in the JavaScript engine to reduce compilation overhead. In order to reuse JIT compiled code, problems for runtime dependency in JIT compiled binary code must be resolved. We used a direct binary code patching method on Squirrel FisheXtreme (SFX) JavaScript engine of WebKit for the experiment. Through the experiment, we showed that the total compilation time of the modified SFX JavaScript engine was slightly increased up to 9.4% by saving codes, but the time was reduced up to 49%, averagely 44%, depending on web services when the code was reused.

Key words: JavaScript engine, JIT compiler, binary code patching
Communicated by: M. Gaedke & T. Tokuda

1 Introduction

The purpose of JavaScript was to implement dynamic effects to the client of web pages, and it attracted the attention of web developers and designers at the beginning. In the mid-90s, it also caught other developers' attention, because a new technique that can link directly to servers such as AJAX (Asynchronous JavaScript and XML) has been developed and because its application area has been extended [1]. Now, JavaScript has become the most frequently used web language. It is used in more than 90% of the main web pages in the world [2]. However, JavaScript is basically a script language whose execution speed is slow. As it is widely used in many application areas, a new demand for the

engine that can accelerate the execution speed of JavaScript is increased. Therefore, JIT compile technique is used for increasing the execution speed of JavaScript engines [3,4,9,10].

JIT compiler compiles only the source codes of actually executing parts of programs at run-time. Because the compiled code can be used again later during the run-time, it can eliminate the duplicated compilations whenever it is required. On the other hand, JIT compiler deteriorates the startup speed compared to the interpreter because it requires conversion before executing programs [5]. Numerous studies have attempted to reduce the startup speed of JIT compilation technique, especially in JVM (Java Virtual Machine), in which JIT compile technique has been embedded. Adaptive Compilation is the most typical technique, in which interpreter is combined with JIT compiler, or various compilers are arranged in order, whose optimization levels are different. It uses several engines to speed up the startup time of program by dispersing initial overhead [6]. Most of the adaptive compilation techniques are used in desktops or server JVMs. However, it is difficult to use the adaptive technique in the embedded systems whose computing resources are limited, since it is required to combine interpreter with multi stage JIT compiler [7].

Consequently, there are many attempts to reduce the startup speed of virtual machines with a code reuse method on embedded systems. Code reuse method is a technique of reusing complied codes continuously; the code which has been compiled once is saved and becomes reusable. JVM compiles a bytecode, saves it as a native code at the loading time, and then loads and runs the native code when it executes a program. The prepared compiled binary code only needs to be converted to the executable form on virtual machine instead of recompiling the byte code again, so it can minimize startup overhead up to its interpreter level. Also, it is possible to re-optimize code with code usage pattern because compiled code is saved in advance [9, 10]. However, it is difficult to apply the code reuse method with the characteristics of static compiler to applications with the feature of dynamic language.

While many JavaScript engines have their own adapted JIT compilers to increase the execution speed, the code optimization techniques still need improvement [3, 9, 10, 11]. Due to the early stage of JavaScript, JavaScript developers have developed JavaScript engines with low performance. So, the start-up speed and optimization were not their concerns. However, since web technology has been improved, there were more complicated applications that adapted more complicated optimizing techniques and that caused JavaScript to decrease the startup speed just like JVM. In order to increase the speed without decreasing the startup speed of program in JavaScript engine that JIT complier is applied, this paper presents the reuse of JIT compiled code based on the binary code patching in JavaScript engine. This method can decrease the number of times being compiled because it can reuse duplicated code in JavaScript. In addition, it can track the frequency in use of compiled binary code, which leads to analyze program patterns and then helps in optimizing the code.

We implemented the idea to reuse JIT compiled code with SFX (SquirrelFish Extreme) JavaScript engine of WebKit. SFX engine is one of latest JavaScript Engine with JIT compiler. To verify the effect of the proposed method, we measured the execution time and frequency of the methods that apply compiled overheads for the major web services. In order to reuse JIT compiled code, we modified SFX engine to save JIT compiled code that separates the repository. If the JIT compiled method is called again, the corresponding binary code can be loaded directly. In order to support this process more effectively, special binary formats and repository of saved code in binary format are also required. However, we will discuss this subject in a separate paper [22].

This paper is organized as follows: in section 2, we describe the outlines of native code reuse methods in virtual machines based on JVM applications and a structural feature of JavaScript engine based on SFX engine of WebKit. Then, we address the operations of JavaScript engine in section 3, such as the process of reusing binary code, and the necessary code patching process when a binary code is loaded. Section 4 describes the details of a layout and an execution of modified JavaScript engine. In section 5, we show that this proposed method of reusing code can reduce compilation overhead. And finally, we indicate conclusions and further research in section 6.

2 JVM and Binary code reuse

The dynamic feature of Java language prevents reusing binary code on JVM environments. Although Java's dynamic class loading can change inheritance structure at runtime, it is impossible to change the structure of code compiled by static compiler. JVMs, which reuse binary code, suggest several methods to solve this problem.

Keller and Hölzle suggested BCA (Binary Component Adaptation), which executed intermediate codes after converting them to the appropriate binary codes form for run-time environment in loading time, instead of using native codes directly. This technique indicates that previously compiled binary codes could be used in new JVM run-time environment after the original codes were converted at loading time [12].

Conte, Trick, Gyllehaal, and Hwu used the feature that Java applications and Applets share a space when they are activated. This feature can reduce the invalidation of compiled codes by effectively managing code cache. This also verified that the process of internal code converting in JVM's interpreting process could be reduced [13]. Even though both research results show a possibility that reusing binary codes can be applied to JVM, they did not aim to use the actual native codes as a reusing target.

The research of reusing native code has been preceded in QuickSilver's quasi-static compiler system by Serrano, Bordawekar, Midkiff, and Gupta in IBM TJ Watson's research laboratory [14]. Similar to the Keller and Hölzle's method, QuickSilver's quasi-static compiler compiled Java bytecode for an intermediate code in advance. Then, QuickSilver's run-time system modified again the intermediate code in an appropriate format for run-time by code patching before it was executed. Since quasi-static compiler offered advanced code optimization, which JIT compiler couldn't offer, they obtained code with better performance speed. However, this method couldn't be used in standard Java environments because their application was limited.

On the other hand, some embedded systems could execute codes directly on ROM to save system memory (RAM), and code patching techniques for modifying program couldn't be applied in this condition. Joisha, Midkiff, Serrano, and Gupta suggested a symbol mapping method using indirect tables to overcome the limitation of Quicksilver's code patching [7]. If a new code accesses to a run-time function, it refers to a specific location in an indirect table instead of an address of function. In addition, whenever run-time changes, corresponding functions or addresses of objects are updated in the table to solve dependency problems. Accessing overhead through the table reduced its performance by 1~7%.

Meanwhile, Hong et al. suggested c-AOTC (Client Ahead-of-time Compilation) technique, which compiles Java code in advance on the server, then downloads and executes it on the client [8]. The dependence problem is solved by using the indirect table mapping technique similar to Joisha's method, but they processed exceptional handling with an interpreter, since this technique is based on Sun's Hotspot VM.

Those studies on JVM didn't reuse compiled JIT code directly, but they supported Java's dynamic run-time environment with static compiler methods. Therefore, they could adapt to the change of run-time difference from compilation time by converting compiled binary codes in advance at loading or by using an indirect table which could access to program information. Their results focused on improving JVM performance in embedded conditions, and the results of IBM research were applied to IBM J9 commercial JVM's embedded version [15].

3 Operation of JavaScript Engine

3.1 JIT Compiler of JavaScript Engine

Since JavaScript supports dynamic type, it can't confirm variables' types at compiling time, and it supports features of functional language such as Closures and Higher-order function like Scheme and Ocaml. Also, it supports prototype-based inheritance which can change an inheritance structure among objects arbitrarily at run time. It is impossible to obtain information needed for code optimization using a static compiler due to the dynamic feature of JavaScript language.

Gal, Probst, and Franz suggested 'Tracing JIT' technique to optimize code after analyzing program's execution pattern and collecting information for code optimization [16, 17]. This technique was applied to TraceMonkey JavaScript engine, which was used in Firefox 3.5 with the project of Mozilla foundation. Nevertheless, 'Tracing JIT' technique was ineffective because the JavaScript engine needed to be switched back and forth between the interpreter and the compiled machine code whenever it reached a certain condition. To make 'Tracing JIT' effective, source codes should be formed in complicated logic so it could cause some computational load. However, most JavaScript codes used in web pages turned out that switching back and forth between interpreter and compiled machine code happened a lot more than expected. Therefore expected execution speed can't be obtained with 'Tracing JIT '[11, 18].

Recent JIT compilers of Apple's SFX or Google's V8 JavaScript engine were designed to fit in environments for small codes. These engines compile JavaScript applications by unit of methods. They considered general features of programs used on web pages so they didn't analyze complicated codes for traditional code optimization; they optimized them in intuitional way instead. Although they suggested a simple method of optimization like Inline Cache, which could reduce the cost of access attribute or the method of object, or Constant Folding at bytecode level, their performance was better than TraceMonkey's [10, 11, 19, 20, 21].

3.2 Dynamic Characteristics of JavaScript Engine

JavaScript engine executes a program throughout 4 steps: Load, Parse, Transform, and Execute. First, 'loading source codes' and 'managing loaded source codes' are referred to 'Load' step. Then,

'confirming that code is suitable' is 'Parse' step before it moves on to 'Transform' step, which transforms an appropriate form in order to execute. 'Transform' process can be different depending on engine's executing method. SpiderMonkey engine compiles the source codes as inherent bytecodes and executes the compiled code by a bytecode interpreter, but SFX engine takes 2 steps to execute: it compiles source codes as byte codes first, and then converts them to native codes before execution. Google's V8 engine compiles and executes source codes as native codes directly. SFX and V8 engine execute native code, which can achieve higher speed than SpiderMonkey engine, since it is executed by a byte code interpreter [10].

JavaScript supports interactive programming like other general Script languages, so it does not have main function unlike C or Java. It also creates programs randomly at the runtime through evaluation function (eval()). In order to support that effectively, JavaScript engine needs to classify loaded programs into three execution code types: global, function, and evaluation code type.

JavaScript does not have a standard about virtual machines such as Java bytecode, and it should keep language compatibility instead. All execution units are dynamically generated from source codes at run time in order to support JavaScript semantics. So, JavaScript engine needs to maintain an original program source code. Thus, it is impossible to use a code identifier based on binary code files using JVM class files, and impossible to manage compiled binary codes in class file units similar to Quicksilver, AOT, and BCA.

Executable binary codes of JavaScript should be managed according to their execution forms in JavaScript engine's internal processing units such as global code, function code, and evaluation code. We will discuss more about this subject in our separate paper [22].

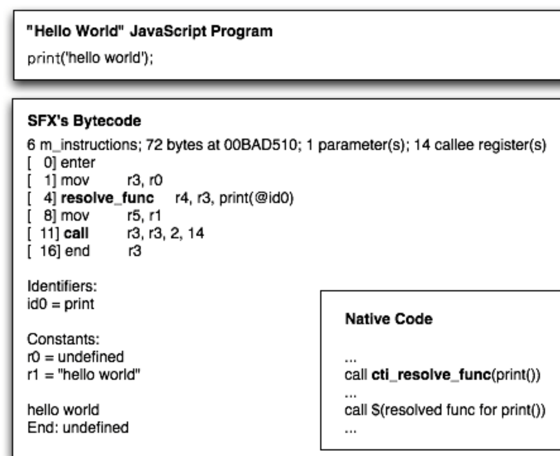

Figure 1 A process of code compilation in SFX engine

3.3 Runtime and Data Dependency

Figure 1 shows a process of converting JavaScript program into SFX engine. A simple code that prints out letters on the screen is converted to a bytecode that is divided into identifiers, constants, and

actually executable bytecodes. ‘Resolve-func’ is an internal support function used in finding and printing out print() function on the screen. Then ‘resolve-func’ will find the specified function in the current namespace, and it will inform its location. This command is written in C language for convenience of implementation due to a difficulty of writing in assembler. Most commands except simple operations like arithmetic and memory operations are written in C/C++ functions–cti_resolve_func(). So, it is easy to call the function in assembler because it is implemented with C ABI (Abstract Binary Interface). But native codes with completed links use addresses instead of function’s symbol when a run time function is called.

The address of function applied to native codes is valid only for the time when code is created. If binary codes compiled previously are loaded, they will not be executed because the runtime function address in code is not valid. This is called runtime dependency of JIT compiled native code. Before reusing native codes created previously, runtime dependency on runtime function must be resolved.

In addition, JavaScript engine manages with memory manager not only objects, arrays, variables, functions, and alias of objects, but also call stack of functions and even native code when a program is executed. SFX engine figures out address spaces that are created by JavaScript objects when compiling bytecode, and assigns them in its runtime. An assigned area is used for JIT compiled native code to access the runtime data, so the address should be changed into the valid address that can be accessed by an object. This is called data dependency of JIT compiled native code. Numerous studies of JVM mostly solved these problems using ‘code patching’ or ‘indirect tables.’ In this paper, we used binary code patching method to minimize changes in SFX engine’s run time structure.

4 Design and Implementation

In this paper, we used SFX engine to reuse JIT compiled code. Although Google’s V8 engine was developed for web rendering engine of WebKit, V8 was designed to convert source codes to native codes directly through AST (Abstract Syntax Tree) without generating intermediate bytecodes, so it was difficult to separate a part of program that creates native codes.

4.1 The structure of Code Reuse Module

Figure 2 shows the whole structure of JavaScript engine that has a code reuse module. In order to understand easily, the code reuse module (Store, Reload, and Purify) is added to JavaScript engine’s regular procedure of Load, Parse, Transform, and Execute.

The first step of code reuse module is ‘Store’. This step stores compiled native codes of a method in assigned repository. Also, binary codes are stored according to the binary format specification with additional information needed for later loading.

Before executing program codes, JavaScript engine examines the existence of previously compiled binary codes. If these compiled binary codes are in existence, they should be reloaded to the engine; and if they don’t, they should be compiled as new codes. In order to find the existence of previously compiled binary codes, a hash key is created by using the source codes and is checked with other hash keys to confirm that the source codes are not the same. When binary codes are loaded, they need to renew the information of binary code usage in binary repository. Finally, loaded binary codes in the

engine should be processed through 'Purify' step, which renews runtime information for new runtime environment, and registers to the code block before execution.

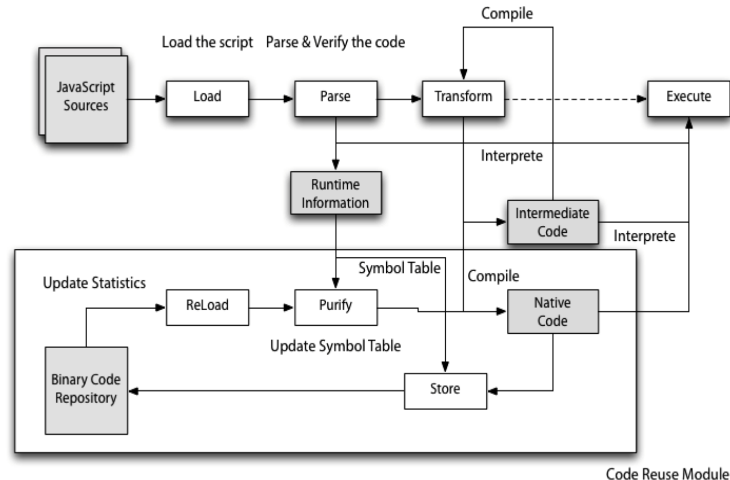

Figure 2 JavaScript engine's structure with code reuse module

4.2 Binary Code Patching

'Code purify' is a process that revises JavaScript engine's runtime dependency in loaded native codes. SFX engine manages compiled native codes separately so it uses 'Executable Pool.' Modified SFX engine also uses 'Executable Pool' to manage loaded codes for reusing, which needs to process 'Code purify' step.

Memory footprint refers to the amount of main memory that uses a program or references while it is running. A code patching method needs to duplicate the target code section for patching the original target code. Because, virtual memory system in modern operating systems mostly adopts copy-on-write policy to effectively use system memory. Therefore, code patching method increases the memory footprint. On the other hand, a method of using indirect tables doesn't need to increase the memory foot print. In addition, the method of using indirect tables has an advantage that can share codes as ROM image forms, so it is very advantageous on embedded system. Meanwhile, the method of code patching has strength in easy execution because it doesn't need to change the runtime structure of existing engines, while the method of using indirect tables needs to create execution codes as a form for accessing indirect tables, and therefore to modify the runtime structure of engines [9].

Since SFX engine executes compilers and programs in order, there is no synchronization problem between compilers and programs, which can be easily seen in an adaptive compilation method. Although the problem of memory footprint, which is a disadvantage of code patching method, is an important problem in systems with limited memory like embedded systems, the problem is not considered in this paper, in which we primarily focus to realize a basic environment needed for reusing codes.

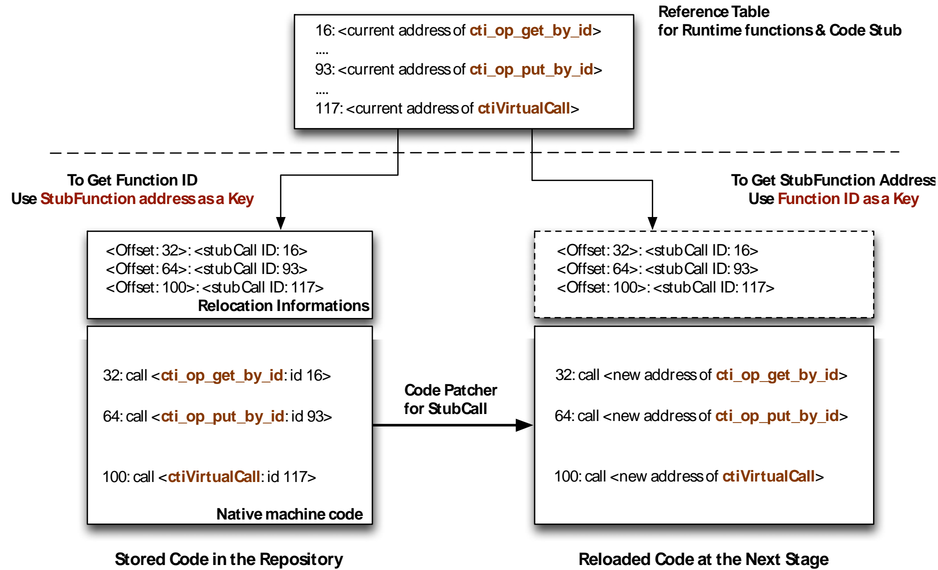

Figure 3 A code patching process of C runtime function and global code stub address

Figure 3 shows a process that our modified SFX JavaScript engine patches the addresses of C runtime functions and global code stubs. The reference table for runtime functions and code stubs is created in the JavaScript engine. This is used to get the runtime stub function ID by its address or the stub function's address by its ID. Therefore, in the table, all runtime functions and code stubs are stored with their own unique ID and the address they are currently loaded in the engine. This reference table is renewed whenever the JavaScript engine is executed because the runtime functions and code stubs aren't loaded to the same address.

Now, it is assumed that JIT compilation is completed. The code reuse module can get the compiled native codes and it can generate binary files which will be reused later if the method is called again. The generated binary file has a relocation information table for runtime functions and compiled native codes. A relocation information in the table is formed with the file offset and function's unique ID (ex, <file offset: function ID> to indicate what function or code stub is called at this position. The file offset indicates the called position of a runtime function or a code stub. It is possible to get the target runtime function address, and then get the target function ID from the reference table.

Finally, when the saved code is called in an application, the engine loads the binary file and the loaded binary file is passed to the code purifier in the code reuse module. The code purifier reads a relocation information entry in the relocation table and it finds a call position using offsets in the entry. Then, the code purifier rewrites the call operation with a new target runtime function or code stub's address. If call operations are patched with new target addresses, then the code purifier needs to patch runtime data reference.

Figure 4 shows a code patching process of runtime data reference from loaded binary codes. This is the same concept with patching runtime function addresses because some operations directly refer addresses of runtime data in figure 3. The reference table in figure 4 hasn't direct addresses of data, but

has runtime information tables. This reference table is created at loading time of JavaScript application.

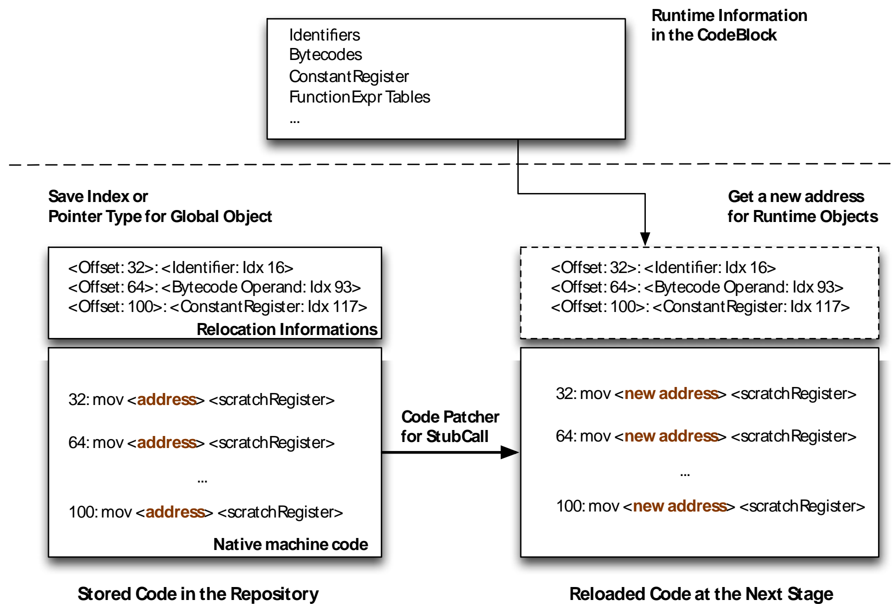

Figure 4 A code patching process of runtime data reference

Another relocation information table should be created to indicate what symbols are needed to relocated. A relocation information entry is formed with the file offset of runtime data accessing code, type of runtime data, index, or constant value according to its form. Then it saves to the binary code repository.

Finally, the code reuse module loads a binary file from the binary code repository and purifies the native code in the binary file with its relocation information. Code 'purifier' patches an address or a value of the object with newly allocated area or assigned values in SFX engine's runtime. A code patching method depends on data type of the object. The reference table has a data index, which is used in bytecodes. As shown in figure 4, <Offset:32>:<Identifier:Idx 16> indicates that data type is 'Identifier,' and index is '16' in the identifier table. SFX engine manages the runtime information of the program such as Identifier, Constant, Bytecode, Function Expression, and so on. If the runtime information can be retrieved from bytecodes, then it is possible to use the bytecode index as a data index, and a data type is named as 'Bytecodes.' If the runtime information can be retrieved from an independent information table such as 'Identifier' or 'Constant Register,' then an independent data index should be used.

5 Experiments and Results

We used XCode 3.1 of Mac OS X 10.6 Snow Leopard and script of WebKit. The core module is developed in C++ language. And WebKit library added with code reuse function is linked and used with the Safari 5 of Mac OS X. Also, Intel’s I5-720M Mobile version CPU (Clock 2.4GHz, Cache

3MB, Dual Core), 8GB (DDR3-1066MHz) memory, and Western Digital 2.5” 500GB (5400rpm, Cache 16MB) hard disk are used.

Since it is not possible to compare the compilation speed with JavaScript benchmark tool of WebKit, SFX engine’s compiling time is measured for accurate performance measurement. Compiling time is classified into three cases: when it saves code in repository, when it uses the saved code, and when it doesn’t reuse the code. Compiling time for each case is measured and compared with non-reusing codes. We measured the compiling time until the page loading of each web service is finished.

Table 1 Web Services for Experiment (Oct. 2010)

Web Services	Compile Count	URL
Google Wave	5,527	wave.google.com
Google Reader	911	reader.google.com
Google Gmail	3,778	gmail.google.com
Google Word	4,414	docs.google.com
Google Spreadsheet	4,955	docs.google.com
Twitter	1,024	twitter.com
Facebook	990	facebook.com

Table 1 shows web services with their compile counts in this experiment. Compiling counts shows the number of compiled method from the beginning of web service to the end of page loading, which is the same with the total number of methods until each web service’s main page is loaded. However, the number is not fixed whenever web pages are loaded. So, we used the average number of compiled method after repeating 10 times in table 1.

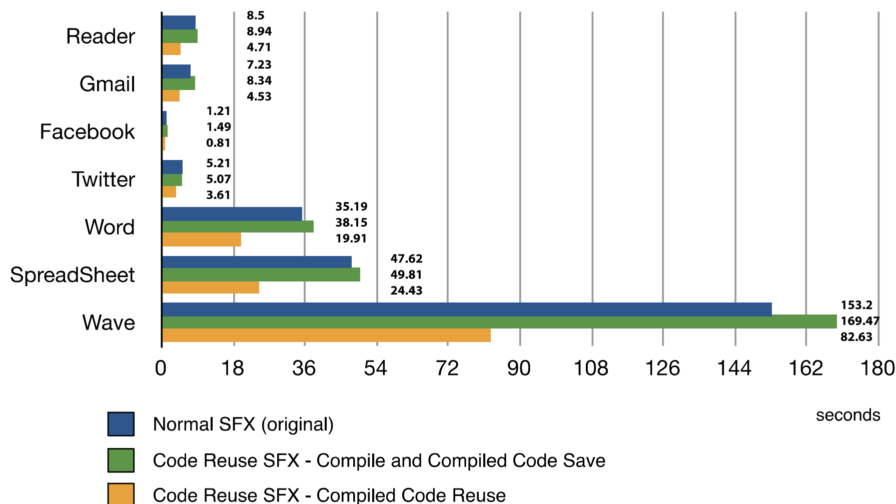

Figure 5 Comparison of Total Compiling Time

Figure 5 shows the total compiling time in a second. Google Wave compiles 5,527 methods. It took 153 seconds in SFX engine when the code reuse method was not applied, 169 seconds in the engine that reuses codes when it saved binary codes, and 83 seconds when the code reuse method was applied. Consequently, we could save 46% of compiling time in a comparison with the time when codes were not reused. As in figure 5, each web service has different performance. FaceBook reduced 31% of time at a minimum, Google SpreadSheet reduced 51% at a maximum, and the average reduced 41% of compiling time. Even though extra time is required for storing the codes at a saving step, which is overhead of the method, it is possible to save much more time with the code reuse method.

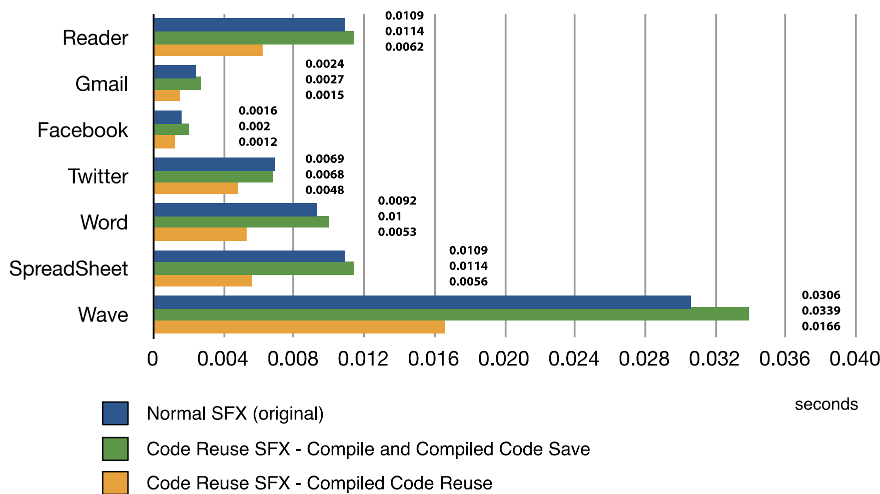

Figure 6 Comparison of Average Compiling Time of Method by Web Service

Figure 6 is a comparison of average compiling time of methods in each web service. Google Reader service compiles about 911 methods until it loads the main page, while Google's Gmail service compiles more than 3,778 methods. Nevertheless, the result shows that Google Reader takes a longer compiling time, which means that each method of Google Reader service is relatively bigger and more complicated than Gmail service. If a method is complicated, its compiling time will be increased, but the code reuse method proves that it reduces the required time and has better effect for the method.

The average compilation time in figure 6 shows about 49% of improvement, which is quite similar to the results of total compiling time in figure 5. Especially, services which have long compiling time such as Google's Reader, Word, SpreadSheet, and Wave are improved more effectively. When the longest compilation time of each service is compared, the compilation speed is reduced up to 63% and averagely 52%. Moreover, the methods whose code size is big have longer compiling time. However, the improved rate of a maximum compiling time is decreased up to 41% because its overhead also is increased for additional operations such as input/output of files, file identification policies based on MD5 hash key, and code patching if a code is too big like Google Wave.

6 Conclusions and Future Works

The purpose of JavaScript was to implement dynamic effects on a client of web pages, and it attracted the attention of web developers and designers in the beginning. Now, JavaScript has become the most frequently used web language. It is used in more than 90% of the main web pages in the world. However, JavaScript is basically a slow performing script language. As it is widely used in many application areas, a new demand for the engine that can accelerate the execution speed of JavaScript is increased. Therefore, JIT compile technique is applied to increase the execution speed of JavaScript engines.

As JavaScript is used in many applications and widens its application areas, the size and complexity of programs are increased. Therefore, there are lots of demands to improve JavaScript engine's performance. This paper presents an improvement of JavaScript engine by reducing JIT compiler's dynamic load using JIT compiler technique.

However, JIT compiler can increase the virtual machine's execution speed, while its compiling overhead deteriorates user's reaction speed. There are numerous studies conducted for increasing JIT compiler's reaction speed based on the reusing code method in JVM environments. However, our attempt to reuse JIT compiled code in JavaScript engine is tried for the first time.

JIT compiler in SFX JavaScript engine has dynamic language features, and it doesn't perform optimization in order to increase the execution speed while not to reduce the speed of reaction. In the experiment, its compiling time is reduced up to 49%, averagely 44%, keeping the speed of SFX JavaScript engine, and each method's average compiling time is also reduced up to similar rate. In addition, compiling time for large size of programs is reduced up to 63%. On the other hand, the reusing code method slows down the compiling performance for small programs due to its overhead of saving compiled code. Therefore, the code reuse method has advantages in programs that have many big sizes functions, while it has disadvantages in programs that have divided with small size functions.

Most parts of the suggested method in this paper are designed for WebKit's SFX JavaScript engine, so it is required to establish a standard model for other JavaScript engines in order to reuse more generalized JavaScript code. Moreover, reusing JavaScript code can be applied to the runtime profiling of web application, and we hope that this can be used for the code optimization method of JIT compiler. Finally, if this method is applied to web caching and operated as a local system unit or network group unit, it will be more effective by reducing overheads for transforming common codes.

Acknowledgements

The authors wish to acknowledge the collaborative funding support from the Ministry of Knowledge Economy (MKE), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) with grant no. NIPA-2011-C1090-1121-0010, and from the National Research Foundation of Korea (NRF), Korea, under Basic Science Research Program with grant no. 2010-0025831.

References

1. Wikipedia, AJAX (programming), from http://en.wikipedia.org/wiki/Ajax_(programming), 2010.
2. JavaScript Usage Statistics, from http://trends.builtwith.com/docinfo/Javascript, 2010.
3. Boerop, R. The Future of JavaScript Engines: Replace Them with JavaScript Compiler, from http://shorestreet.com/node/43, 2009.
4. JavaScript: TraceMonkey, from https://wiki.mozilla.org/JavaScript:TraceMonkey.
5. Aycock, J. A brief history of just-in-time, ACM Computing Surveys (CSUR), 35(2), pp.97-113, 2003.
6. Hölzle, U. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming, Computer Science, Stanford University, Doctor of Philosophy: 181, 1994.
7. Joisha, P., Midkiff, S., Serrano, M. and Gupta, M. A framework for efficient reuse of binary code in Java, The 15th International Conference on Supercomputing, Sorrento, Italy, ACM, 2001.
8. Hong, S., Kim, J., Shin, J., Moon, S., Oh, H., Lee, J. and Choi, J. Java Client Ahead-of-Time Compiler for Embedded Systems, Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp.63-72, San Diego, California, USA, 2007.
9. Google, V8 JavaScript Engine, from http://code.google.com/intl/ko/apis/v8/intro.html, 2010.
10. Stachowiak, M. Introducing SquirrelFish Extreme, from http://webkit.org/blog/214/introducing-squirrelfish-extreme/, 2008.
11. Blizzard, C. Improving JavaScript performance with JägerMonkey, from http://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/, 2010.
12. Keller R. and Hölzle, U. Binary Component Adaptation, ECOOP'98 - Object Oriented Programming, Berlin/Heidelberg, Germany, Springer, 1997.
13. Conte, M., Trick, A., Gyllenhaal, J. and Hwu, W. A Study of Code Reuse and Sharing Characteristics of Java Applications, WWC'98, Dallas, Texas, IEEE, 1998.
14. Serrano, M., Bordawekar, R., Midkiff, S. and Gupta, M. Quicksilver: a quasi-static compiler for Java, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota, USA, ACM, 2000.
15. Wikipedia, IBM J9, from http://en.wikipedia.org/wiki/IBM_J9, 2010.
16. Gal, A., Probst, C. and Franz, M. Hotpath VM: An Efficient JIT Compiler for Resource-constrained Devices, VEE 06, Ottawa, Canada, ACM, 2006.
17. Mozilla, Tracing JIT, from https://developer.mozilla.org/En/SpiderMonkey/Internals/Tracing_JIT, 2009.
18. Chang, M. etc, Tracing for web 3.0: trace compilation for the next generation web applications, VEE 09, Washington, DC, USA, ACM, 2009.
19. Almaer, D. V8 Internals by Kevin Millikin, from http://ajaxian.com/archives/v8-internals,2008.
20. Ager, M. Google I/O 2009: V8 Internals: Building a High Performance JavaScript Engine, from http://www.bestechvideos.com/-2009/06/04/google-i-o-2009-v8-internals-building-a-high-performance-javascript-engine, 2009.
21. Nikkel Electronics, Why Is the New Google V8 Engine So Fast? from http://techon.nikkeibp.co.jp/article/HONSHI/20090106/163617/, 2009.
22. Jeun, S. and Choi, J. Repository Design for Reusing Binary JavaScript Code, (in preparation).