Web Application Migration with Closure Reconstruction

Jin-woo Kwon
Seoul National University
301-819, 1 Gwanak-ro, Gwanak-gu
Seoul, Republic of Korea
jwkwon@altair.snu.ac.kr

Soo-Mook Moon
Seoul National University
301-819, 1 Gwanak-ro, Gwanak-gu
Seoul, Republic of Korea
smoon@snu.ac.kr

ABSTRACT

Due to its high portability and simplicity, web application (app) based on HTML/JavaScript/CSS has been widely used for various smart-device platforms. To take advantage of its wide platform pool, a new idea called *app migration* has been proposed for the web platform. Web app migration is a framework to serialize a web app running on a device and restore it in another device to continue its execution. In JavaScript semantics, one of the language features that does not allow easy app migration is a closure. A JavaScript function can access variables defined in its outer function even if the execution of the outer function is terminated. It is allowed because the inner function is created as a closure such that it contains the outer function's environment. This feature is widely used in web app development because it is the most common way to implement data encapsulation in web programming. Closures are not easy to serialize because environments can be shared by a number of closures and environments can be created in a nested way. In this paper, we propose a novel approach to fully serialize closures. We created mechanisms to extract information from a closure's environment through the JavaScript engine and to serialize the information in a proper order so that the original relationship between closures and environments can be restored properly. We implemented our mechanism on the WebKit browser and successfully migrated Octane benchmarks and seven real web apps which heavily exploit closures. We also show that our mechanism works correctly even for some extreme, closure-heavy cases.

Keywords

Web application, App migration, JavaScript, Closure

1. INTRODUCTION

The usage of web programming languages (HTML, JavaScript, CSS) is once considered to be restricted to creating simple web documents due to their poor performance. However, as the performances of devices grow rapidly and many optimizations are implemented on the web browser engines, the performance issue has diminished, and the advantages of web programming languages are exposed. One of the advantages is its great portability. Nowadays web browsers are used widely in various devices such as smartphones, smart TVs, tablet PCs and IoT devices. JavaScript has become one of the most popular programming languages, and one evidence is that JavaScript has been the most frequently used programming language on GitHub.com for recent three years (2013-2015) [9].

As many users own multiple devices embedded with a web browser, we can think of a new user experience called *web app migration*. A web app migration is a framework to serialize a running web app's state and restore it in other devices to give a user an experience of seamless execution. For example, we can migrate a game app being played on a smartphone to a smart TV and continue to play with it. Previous researches on web app migration have been proposed [2, 11, 13], and they could migrate some simple web apps by serializing a document object model (DOM) tree and JavaScript objects into text formatted files. However, their approach to serializing *closures* is not supported [2] or supported in a minimal range [11, 13], limiting the capabilities of their works.

A closure is a function that is bound to an environment. When a function is declared in another function, by name binding rules, the inner function can access variables in the outer function. In such case, the inner function is created as a closure; that is, the environment of the outer function is bound to the inner function. In other languages, when a function's execution is finished, its local variables are destroyed. However, in JavaScript, those local variables that are used in the inner function are not destroyed, and they can be reached from the inner function even after the outer function's call is finished. Therefore, the execution engine should keep the environment that contains the local variables defined in the outer function so that the inner function can properly access the free variables afterwards.

In web app migration frameworks, supporting closure is important because they are frequently used in the real world. JavaScript does not have encapsulation model like 'private' field of classes in C++, but many web developers need data encapsulation. To generate an encapsulated variable, it is a common way in JavaScript to use closures. The variables in a closure's environment are only accessible from the closure, so the variables in the environment are completely encapsulated.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052572

Despite the frequent usage of closures, it is not easy to serialize and restore them. The advantage of closures which can encapsulate variables turned out to be an obstacle to the web app migration framework. In order to serialize variables in a closure's environment, we need to access the environment, but in JavaScript layer, it is restricted for other functions to do so. Also, the relationship between functions and environments can be very complicated as there can be multiple closures sharing an environment or closures being defined in multiple layers of environments (nested environments). Serializing such complicated situation is challenging because we need to serialize not only the variables in the environments but also the complicated relationships between the environments and closures.

In this paper, we propose a web app migration with closure reconstruction. We studied the specification of JavaScript language thoroughly and designed our way to extract closure-related information hidden in the JavaScript engine. With the information, we developed a framework to construct the relationship between the closures and to generate the restoration codes of environments and functions properly.

The contribution of this paper is as follows:

- We analyzed the execution model of JavaScript thoroughly and verified all the states we must consider to serialize a closure's state completely.
- We modelled the closure reconstruction problem as a building of tree-like data structure, and we implemented a code generation system to generate state restoration codes based on the information aggregated in the tree.
- We tested our work with seven real world web apps that use closures heavily, and we also tested our work with Octane benchmarks. We have successfully serialized and restored the web apps and benchmarks within acceptable overheads.

The rest of the paper consists of as follows. Section 2 analyzes the execution model of JavaScript thoroughly. Section 3 explains previous works in this area and their limitations. Section 4 introduces challenges on serializing closures. Section 5 shows our design and implementations, and the evaluation is shown in Section 6. Related works are introduced in Section 7, and the summary follows in Section 8.

2. EXECUTION MODEL OF JAVASCRIPT

In order to serialize the states of JavaScript closures, we need to understand the execution model thoroughly. We use the concept and terminology of the model based on the ECMAScript language specification of ECMA-262 5.1 edition [6]. We decided to analyze ECMAScript 5.1 edition because it has been a standard for long period recently (2009–2015), so most of web apps still follow the rule. When a JavaScript program is executed, a global execution context (global EC) is created by default. When a JavaScript function is invoked, an execution context (EC) is created and the control flow is moved to the callee. While the execution is on the EC, when another function is invoked, another EC is created again, forming ECs as a stack. In JavaScript specification, EC consists of three elements which are 'this binding', variable environment (VE) and lexical environment (LE). 'this binding' keeps the binding of 'this' keyword in the execution context. If a function is executed as a property of an object, then 'this' keyword refers to the object. VE holds the initial states of bindings of variables in current EC, and LE holds the current states of bindings of variables. Initially, LE is just a copy of VE, but the bindings in LE are updated during execution while VE remains unchanged. Since LE represents the 'current' state of execution and VE does not, we do not need the information in VE. Therefore we will omit the existence of VE in the rest of this paper for simplicity. An LE consists of an environment record (ER) and a reference to an outer LE. ER is kind of a table that records the identifier bindings in the current LE. There are two kinds of environment records: declarative ER and object ER. A declarative ER is created at function invocation while an object ER is created at the global LE and during 'with' statements.

```
1  function CreateAccount(initial) {
2      var Account = new Object();
3      var balance = initial;
4      Account.deposit = function(value) {
5          balance += value;
6      }
7      return Account;
8  }
9
10 var myAccount = CreateAccount(0);
11 myAccount.deposit(10);      // balance: 10
```

Figure 1: Example code for closure creation

Each JavaScript function object has an internal property named 'Scope'. When a function is defined, its 'Scope' property saves the reference to the current LE. When the function is invoked, a new EC and LE are created, and the function's 'Scope' value is copied to the new LE's 'outer' property. Therefore, by following the 'Scope' property of a function and 'outer' properties of its LEs, a chain of LEs can be found, which is known as a *scope chain*.

We will see this model with an example code in Figure 1 for deeper understanding. The purpose of the program is to create a bank account, and the program wants to encapsulate the balance of each user so that any malicious code can't manipulate it. JavaScript doesn't have protected field as 'private' keyword in C++ or Java, so the most typical way of hiding variables in JavaScript is to create a function as a closure and hide the variable in the environments. We will see how the variable 'balance' is encapsulated in the example. In line 10, the function 'CreateAccount' is called. The local variable 'balance' is declared in the execution context of the call, and it is supposed to vanish when the execution of the call is done. However, while executing lines 4-6, the program notices that the function 'Account.deposit' is using the variable 'balance' that is positioned in the scope of 'CreateAccount'. Even though the execution of 'CreateAccount' is finished, 'Account.deposit' needs to access the variable 'balance' afterwards, so the JavaScript engine creates an environment that keeps the variable 'balance' and gives an exclusive authority to access the environment to the function 'Account.deposit', making it as a closure. After execution, now the variable 'balance' is only accessible via an invocation of 'Account.deposit', so 'balance' is protected from any other codes.

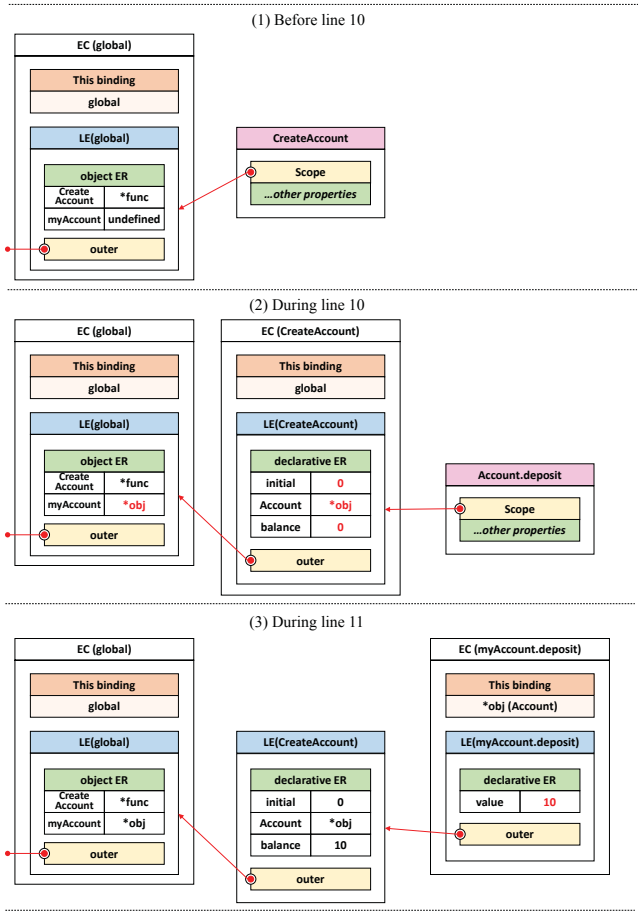

Figure 2: Execution model for code in Figure 1

To see what exactly happens in the JavaScript engine, we will see how EC stacks are generated by the execution of the code snippet. Figure 2 illustrates the generation of EC stack after running the example code. At the beginning of the program, the JavaScript parser scans over the code to gather the information of variables. Then, a default global EC and LE are created with an object ER that contains all the identifiers in the global LE. 'CreateAccount' identifier is declared as a function, so the program instantly knows it, but the variable 'myAccount' is initialized as 'undefined' value because line 10 is not yet executed. In lines 1-8, function 'CreateAccount' is created, and its 'Scope' internal property is set to the global LE. At line 10, when the function 'CreateAccount' is called, a new EC and LE is created, and LE's 'outer' property is set to the function's 'Scope' property (which is the global LE). In lines 4-6, when the function 'Account.deposit' is created, its 'Scope' property is set to the LE of 'CreateAccount'. After line 10, since the execution of the 'CreateAccount' is finished, the LE should be destroyed as well, but 'Account.deposit' function's 'Scope' property is still pointing it so the JavaScript engine decides to preserve it. Therefore, when the function 'myAccount.deposit' is invoked in line 11, the execution can find the variable 'balance' by following 'outer' property of its LE.

From this study on execution model of JavaScript, we conclude that in order to serialize a closure, not only the function itself is needed to be saved but also its internal scopes (environments) are needed to be saved.

```
1 var closure = function () {
2     var balance = 10;
3     return (function(value) {
4         balance += value;
5     });
6 };
7
8 var myAccount = new Object();
9 myAccount.deposit = closure();
```

Figure 3: Restoration code for code in Figure 1

3. PREVIOUS APPROACHES

There has been previous studies on app migration framework [11, 13] that could handle simple closure problems. Lo et al.'s work named Imagen [11] instruments the original source code to access and capture a closure's environment. When the instrumented code is executed, the LE is exposed to other codes, so they can save and restore the closure with the environment. Oh et al.'s work [13] modified the V8 JavaScript engine [5] to add new APIs that can retrieve scope chains in a closure. After retrieving the LE information of a closure, both approaches create a restoration code in a way that a wrapper function wraps the closure so that the wrapper function can restore the LE and bind with the closure. The restoration code on Figure 1 generated by the previous approaches will be Figure 3. The object 'myAccount' is restored in line 8, and the function 'myAccount.deposit' is restored by invoking the function 'closure' in line 9. By executing the function 'closure', a new environment that has variables identical to those of the previous LE (variable 'balance' with value '10') is created and bound to the returning function. Therefore, the function 'myAccount.deposit' is successfully restored as a closure.

Unfortunately, previous approaches cannot restore closures correctly if the situations get any more complicated. Even adding another closure in a LE can be a trouble. For example, lets assume that a codes of Figure 1 are extended as in Figure 4. The added parts are marked in gray. The developer added another function named 'Account.calculate' (line 8) that shares the LE with the 'Account.deposit'. Previous approaches are not able to identify that the LE of 'Account.deposit' and 'Account.calculate' are identical, resulting in two closures restored with two duplicated wrapper functions separately. It is semantically incorrect restoration since the restored closures will be having different LEs.

4. CHALLENGES

In this section, we verify the challenges we faced to restore closures correctly and explain how we resolved each of them.

4.1 Nested environments

Closures can share an LE, and LEs can form a scope chain. In this situation, it is not straightforward to restore closures because the complicated relationship between the closures and the LEs should be restored as well. One example is the code in Figure 4. Figure 5 shows the resulting state of closures and their environments after executing the code in Figure 4. Functions 'Account.deposit', 'Account.calculate',

```
1  function CreateAccount(initial) {
2      var Account = new Object();
3      var balance = initial;
4      Account.deposit = function(value) {
5          balance += value;
6      }
7      Account.rate = 1.05;
8      Account.calculate = function() {
9          balance = balance * this.rate;
10     }
11     with(Math) {
12         Account.round = function() {
13             balance = round(balance);
14         }
15     }
16     return Account;
17 }
18
19 var myAccount = CreateAccount(0);
20 myAccount.deposit(10);     // balance: 10
21 myAccount.calculate();     // balance: 10.5
22 myAccount.round();         // balance: 11
23 var calculateWithNewRate =
24       myAccount.calculate.bind({rate:2});
25 calculateWithNewRate();    // balance: 22
```

Figure 4: Expanded version of the code snippet (added codes are darkened)

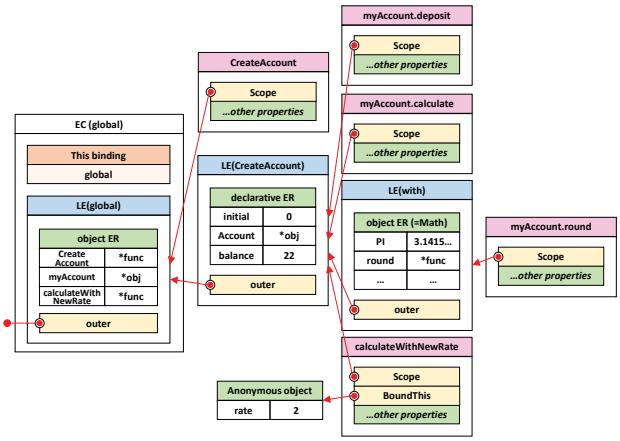

Figure 5: Function object and scope relationship for code in Figure 4

'Account.round' and 'calculateWithNewRate' are sharing the same LE. Moreover, there is a LE created by 'with' statement that 'Account.round' function's 'Scope' property is pointing to, which makes the situation more complicated. The LE created by the 'with' statement will be discussed in the following subsection. In this situation, it is not easy to create a restoration code because it is hard to reconstruct the relationship. The problem is that the scope chain is directed from the function to the global LE, and not the other way around. We can never know if 'Account.deposit' and 'Account.round' is in the same scope until we track the scope chain and compare the LEs. We mapped this nested environment problem into a tree building problem and successfully reconstructed the relationship to generate the complete restoration code.

4.2 'with' statement

In JavaScript, 'with' statement is used to actively add a scope to a scope chain during the execution of its statement body. The given object as an argument to the 'with' statement acts as a scope object, whose properties being the identifiers for the scope. For example, in line 13 of Figure 4 a function 'round' is invoked. In the scope, there is no local variable that matches 'round' identifier, so the execution tracks scope chain to find a variable that matches the identifier. Because line 13 is in the statement body of 'with' statement given 'Math' object as an argument, the execution searches 'round' identifier in the properties of 'Math' object. 'Math' object is a global built-in object that gives many library functions related to calculations, and one of them is 'Math.round' function. Therefore, the execution matches the identifier 'round' to 'Math.round', so the call on 'round' becomes a call on 'Math.round' function. The LE that is created by the 'with' statement contains object ER rather than declarative ER, so the restoration code should be generated differently.

4.3 This binding

This binding is a part of an execution context that guides which value should be taken when 'this' keyword is used in the current context. This binding is decided when an execution context is created according to the context of how a function is invoked, so it is generally irrelevant to a function's state. However, there is an internal property field named 'BoundThis' in a JavaScript function that can force 'this' value in the execution context to be a certain value when it is invoked, regardless of the current context. Since the value of the internal property is also a state of a function, it should be saved as well. 'BoundThis' property is only set when a function is created with '.bind' function.

To explain the challenge with the example in Figure 4, a function 'calculateWithNewRate' (line 23) is created by calling '.bind' function to the function 'myAccount.calculate'. By doing so, the function body of 'myAccount.calculate' is copied to 'calculateWithNewRate'. The copied function has a 'BoundThis' internal property, and its value is set to an object {rate:2}. When 'calculateWithNewRate' is invoked (line 25), the function calculates `balance * this.rate` as if 'myAccount.calculate' is invoked, but 'this' refers to the object {rate:2} rather than the 'Account' object. Therefore, the the resolved rate is 2, doubling the total balance. If we serialize the 'calculateWithNewRate' function, 'BoundThis' property should be restored as well; if not, it will behave differently when it is restored.

5. CLOSURE RECONSTRUCTION

5.1 Overall design

When a web app is launched, a global built-in object named *window* is allocated by default. When a JavaScript variable is declared in the global scope, it is attached as a property to the window object. That is, if 'var a' is declared in the global scope, then it is equivalent to creating 'window.a' property. Therefore, by tracking all the properties of the window object, we can find all the global objects that are currently live in the state. Our serialization framework basically exploits this characteristic. We access all the properties of the window object, and the properties of the found

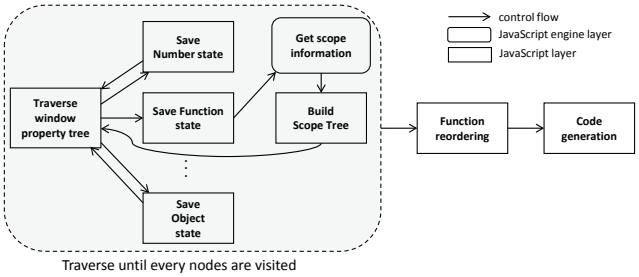

Figure 6: Overall design of our framework

objects, and repeat until every object is accessed. In each access, we serialize each property.

Figure 6 illustrates the overall design. Most parts of the framework are written in JavaScript (rectangle) while some parts are modified in the JavaScript engine (rounded rectangle) to get the scope information. Our framework traverses window object's properties to visit and serialize objects in a tree traversal manner. We named this traverse as window property tree traverse. Figure 7 shows how JavaScript objects can be restored. While traversing, when primitive types (Number, String, Boolean, Null, Undefined) are found, the value can be serialized directly ('x' and 'y'). When an object (including function object) is found, the object is first checked with a reference array ('ref'). A reference array is a unique list of data that the framework manages to check which object is already traversed and serialized. If it exists in the reference array, it means that it is already found and serialized previously, so the framework simply serializes the reference array index number. If it does not exist in the reference array, it is pushed into the reference array. In this way, multiple variables pointing the same object ('z' and 'z_copy') can be restored without duplicated creation.

With this approach, primitive types and objects besides functions can be easily serialized. As we discussed earlier, JavaScript functions may be bound to its environment. Therefore, when a function object is found, we call an API function implemented in the JavaScript engine to retrieve a full scope chain data of the function object. The scope chain data are used to rebuild the scope tree that has a global scope as the root of it. The framework continues traversing objects and their properties until it visits every object in the window property tree. When it is done, we can guarantee that all the scope chains are found and a scope tree is completely rebuilt. Then we check the initialization dependencies between scopes and functions and reorder them in the right order so that all the initialization dependencies can be satisfied. Lastly, when all the steps are completed, the framework generates code from the global scope to the innermost scopes, depending on the scope tree. We will discuss each step in depth in the following subsections.

5.2 Scope information collection

In JavaScript semantics, scope information is hidden from the user layer, and it can't be retrieved from the JavaScript layer. We modified JavaScriptCore (JSC), a JavaScript engine for WebKit browser [15], to implement our custom API to get the scope information. While traversing the window property tree, when a function object is found, it calls the API function to retrieve scope information. Figure 8 shows

(a) Source code

```
1 var x = 0;
2 var y = "Hello";
3 var z = {value:"World"};
4 var z_copy = z;
```

(b) Restoration code

```
1 var ref = new Array();
2 var x = 0;
3 var y = "Hello";
4 ref[0] = new Object();
5 ref[0]["value"] = "World";
6 var z = ref[0];
7 var z_copy = ref[0];
```

Figure 7: Serialization of primitive types and objects

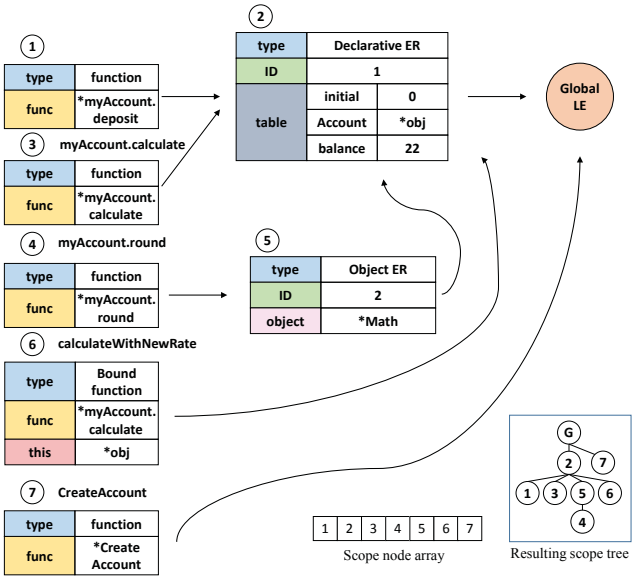

Figure 8: Scope information gathering and scope tree building

the example of scope information gathered from the execution state depictd in Figure 5. Each information is converted to an object, which we call a scope node. When a function object is found, we create a scope node that has 'function' type in the 'type' field and points the function object in the 'func' field (①). Then, we follow the 'Scope' internal property of the function object to find an outer LE. If the LE found is the global LE, then we stop the scope information collection and move on to next function object. Else, the LE found is an environment bound to the function which should be saved, so a scope node representing the environment is created. Its 'type' field is given as 'declarative ER' or 'object ER', depending on the ER type that exists in the LE. In order to identify if the LE is already searched, we retrieve a unique ID (in implementation, we used the memory address of the LE) and save it in the 'ID' field of the scope node. Lastly, 'table' field is generated if the type is a declarative ER (②), and 'object' field is generated if the type is an object ER (⑤). 'table' field saves all the identifiers and values

that exist in a declarative ER, while 'object' field saves the reference to the object that an object ER holds.

In section 3.3, we explained that a bound function needs a special process to restore 'BoundThis' property. When we found a function that has 'BoundThis' property (⑥), we generate a scope node that has special type 'Bound function'. The function's 'func' field is set to the original function object that the bound function is copied from. Also, we give the scope node a 'this' field, which copies the reference saved in the 'BoundThis' property. Both information on 'func' and 'this' field are hidden in the JavaScript semantics, so we modified the JavaScript engine to get the information directly.

5.3 Scope tree building

By collecting scope information, we can generate scope nodes that stand for a function object or a LE. The scope nodes form a chain-like relationships, whose last node is always the global LE. If we collect the scope chains and merge, we can build a tree structure. The global scope becomes a root node, and all the scope nodes can be modelled as a node in a tree. We named the tree as a scope tree. Once a scope tree is built, we can easily traverse the tree and generate restoration codes.

Scope tree building can be done simultaneously with scope information collection. The numbers ① to ⑦ in Figure 8 shows the scope tree building order under the assumption that we found the function objects in the order of 'myAccount.deposit', 'myAccount.calculate', 'myAccount.round', 'calculateWithNew Rate', and 'CreateAccount'. First, we create an empty array named scope node array. When 'myAccount.deposit' is found, ① is created and pushed to the scope node array. Before creating ②, we first search the scope node array so that any scope node that has the same ID exists in the array. Since there is not, ② is created and pushed to the array. When 'myAccount.calculate' is found, ③ is generated and pushed to the scope node array. The same searching process happens when we try to save a LE of 'myAccount.calculate', and this time we find that a scope node with the same ID already exists in the scope node array. Therefore, instead of generating a scope node again, we just let ③ point ②. The same processes are repeated until every function object is found, then we can get the complete scope tree.

5.4 Function reordering

Before generating a restoration code, we need to rearrange the scope tree in the correct order for proper code generation. When generating a restoration code based on the scope tree, we need to decide which child node should be generated first upon many children. Without considering the order, the restoration code might have an error code, such as using a function pointer before the function's declaration. The example code for such case is in Figure 9. The code in (a) is almost the same with the previous examples. If the code in (a) is restored as the code in (b), it would be a wrong restoration because the declaration order of functions is not correct. In (a), the variable 'globalFoo' is defined before line 4, where the variable 'foo' is set to a value saved in 'globalFoo'. Therefore, the variable 'foo' will properly be pointing the function that prints "foo". However, in (b), the restoration of 'myFoo.invokeFoo' is happened before the restoration of 'globalFoo'. Therefore, while executing line 2, the 'globalFoo' value is undefined, and 'foo' will become undefined value as well. As a result, the call to 'myFoo.invokeFoo' in line 10 results in error. To correct this, line 9 should be generated before line 1.

(a) Source code

```
1  var globalFoo = function() {print("foo");}
2  function CreateFoo() {
3      var Foo = new Object();
4      var foo = globalFoo;
5      Foo.invokeFoo = function() {
6          foo();
7      }
8      return Foo;
9  }
10 var myFoo = CreateFoo();
11 myFoo.invokeFoo();       // print: foo
```

(b) **Wrong** restoration code

```
1  function RestoreInvokeFoo() {
2      var foo = globalFoo;
3      return (function() {
4          foo();
5      });
6  }
7  var myFoo = new Object();
8  myFoo.invokeFoo = RestoreInvokeFoo();
9  var globalFoo = function() {print("foo");}
10 myFoo.invokeFoo();       // error
```

Figure 9: **Wrong** restoration without function reordering

To resolve this problem, we do the function reordering stage before the code generation stage. During scope information gathering stage, we collect the dependency information as well. For example, the function 'myFoo.invokeFoo' is dependent on 'globalFoo', meaning that 'globalFoo' must be declared before 'myFoo.invokeFoo'. The dependency information is accumulated in the upward direction in the scope tree. We traverse the scope tree in pre-order depth-first search and arrange the order of children based on the accumulated dependency information. When the arrangement is completed, we can generate the restoration code based on the arranged scope tree.

5.5 Code generation

We traverse the arranged scope tree to generate restoration codes. Pseudo code for code generator is shown in Figure 10. The code generator is called recursively so that every child nodes' code is generated in the scope of its parent node. At the beginning of the code generation stage, the function 'generate' is invoked with the root node (global LE) given as an argument. The function first checks the type of the given node. If it is 'declarative ER', it means that an LE that a function should generate is needed to be created, so we insert a code that can create an anonymous function (line 4). Then, it invokes 'generate' function for all of its children, because children's code should be generated within the current scope. When it is over, we look up the table saved in *node.table* property, and generate regeneration code for every entry exists in the table. If the type of node is 'object ER', it means that we need to create a new 'with' statement. In line 11, we generate a code that can create 'with' statement with *node.object* given as an argument. Then, we traverse the child of the current node as we

Algorithm 1 Code generator

```
1: function GENERATE(node)
2:   switch node.type do
3:     case declarative ER:
4:       code += "(function() {"
5:       for all child in node do
6:         GENERATE(child)
7:       end for
8:       for all var in node.table do
9:         code += stringify(var)
10:      end for
11:      code += "})();"
12:    case object ER:
13:      code += "with (" + node.object + ") {"
14:      for all child in node do
15:        GENERATE(child)
16:      end for
17:      code += "}"
18:    case function:
19:      code += stringify(node.func)
20:    case Bound function:
21:      code += stringify(node.func)+".bind("+node.this+");"
22: end function
```

Figure 10: Pseudo code of the code generator

did before with declarative ER cases. If the type of node is 'function', it is an ordinary function, so we simply generate code to restore its function body. Lastly, if the type of node is 'Bound function', it means that a function is needed to be copied from *node.func* by invoking a 'bind' function, *node.this* given as an argument.

6. EVALUATION

6.1 Experimental Environment

We implemented our work on the open source WebKit browser revision 150000. Our experiment is done on an x86 system with i7-3770 3.40GHz CPU and 16GB memory. To see how it works with mobile devices, we also tested our framework on an embedded board named pandaboard, which uses ARM Cortex-A9 dualcore 1.2GHz CPU with 1GB memory. The behavior of results was similar on both devices.

We tested our work on seven real web apps and a full set of the Octane benchmark 2.0 [12]. All the selected web apps use one of the JavaScript libraries (CentNote, EmotiColor, LightFlip, Phrase, RAM, and Snake use the jQuery library [10], and Tetris uses Prototype library [14]). Five of the apps (CentNote, EmotiColor, LightFlip, Phrase, RAM) are implementations of complex web design and they are crawled from a contest site (`https://a-k-apart.com/`). Other two apps (Snake, Tetris) are implementations of famous games in web languages. We chose such test set for two reasons. First, it is very common for web developers to use at least one of the JavaScript libraries, so the selected test set can show that our system is practical. Second, the JavaScript libraries exploit closures heavily, so by testing them, we can show that our system is robust and stable. The selected web apps generate meaningful state information, and we checked that the state information is restored correctly after restoration. We also chose the Octane benchmark 2.0, because it is the most popular benchmark suite in JavaScript and it tests many features that are specialized for JavaScript language. We serialized the state of each benchmark when the script is completely loaded and is ready to be executed. We ran the benchmark after restoration, and we confirmed that the result is identical to that of original benchmark. We downloaded all the web apps and benchmarks locally so that the network will not affect our results. For both web app tests and benchmark tests, we examined the result five times each and took the arithmetic means.

Table 1: Scope tree results and restoration code size

	Tree Height	LEs	Source code (KB)	Restoration code (KB)
CentNote	4	92	130	402
Emoticolor	4	81	100	392
LightFlip	4	81	97	379
Phrase	4	92	134	3700
RAM	4	116	123	507
Snake	4	80	89	343
Tetris	3	135	109	452
box2d	2	10	240	591
code-load	1	0	129	162
crypto	1	0	63	137
deltablue	1	0	42	75
earley-boyer	2	1	207	676
gbemu	1	0	520	978
mandreel	1	0	4800	5301
navier-stokes	1	0	29	49
pdfjs	4	38	1500	7001
raytrace	1	0	44	83
regexp	1	0	140	159
richards	1	0	32	56
splay	1	0	27	49
typescript	3	118	2500	3911
zlib	1	0	207	225

6.2 Scope tree analysis

We examined the scope tree's height and a number of LEs for each test cases, and the result is shown in Table 1. The result shows that the height of scope trees doesn't exceed 4 in both real web apps and benchmarks. The reason for this is that in JavaScript, scope creation is mostly generated for data encapsulations. Most JavaScript libraries, including jQuery and Prototype which are used in our test apps, create a scope tree to hide its own variables from the user codes so that they can't be manipulated without accessing library APIs. For this purpose, 3 or 4 tree height are enough to hide their variables, so it is rare in the real world to find an app with longer scope tree height. Also, we can see that most benchmark tests don't exploit closure at all, except for four benchmarks ('box2d', 'earley-boyer', 'pdfjs', and 'typescript').

In the third column, the web apps generate more LEs than benchmarks in general. In JavaScript libraries that are employed in the web apps, numerous library functions are internally generated with its unique environment, so a lot of LEs are generated. However in Octane benchmarks, closures are rarely used, so only a few LEs are created.

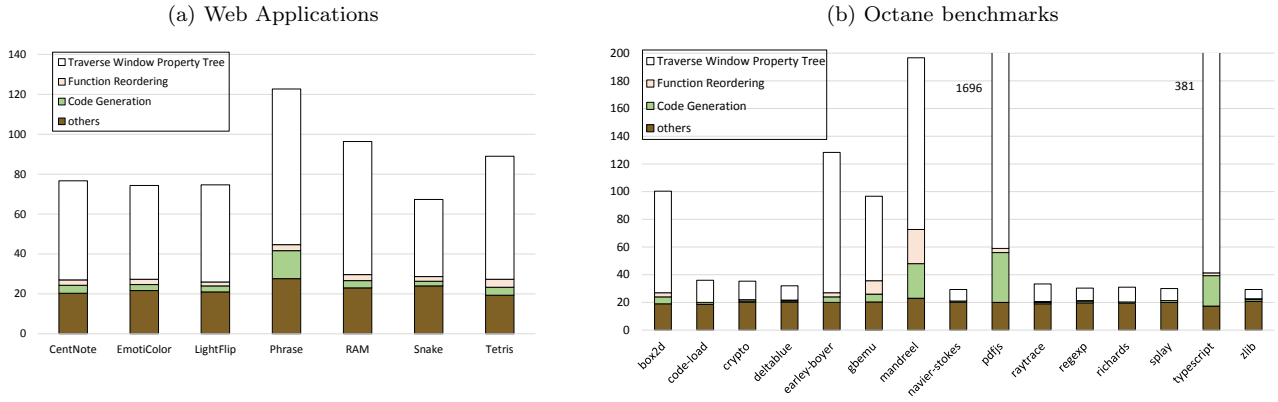

Figure 11: x86 serialization time on web applications and Octane benchmarks (ms)

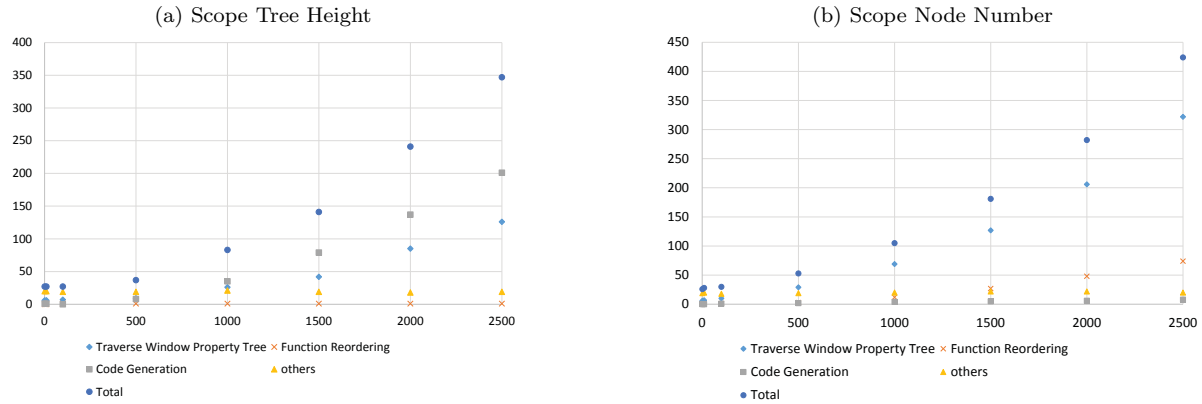

Figure 12: Serialization time on extreme cases

6.3 Restoration code size

The fourth and fifth columns of Table 1 show the source code size and the restoration code size. We can see the trend that the more the LEs are created, the more the restoration code will be. It is a reasonable result because LEs are extra information that didn't exist in the source code, so extra restoration codes are needed. 'Phrase' and 'pdfjs' are two test cases that the size grew abnormally compared to the others. 'Phrase' generates a huge image data during runtime, and the image data are bound to a JavaScript variable, which grew the size of the restoration code. The source code of 'pdjfs' has a function that creates a huge object, and the restoration is done after the creation of the object, so restoring the object grew the size of the restoration code. Both are irrelevant to the number of LEs, so we believe that our closure restoration approach generates the restoration code within an acceptable range of size.

6.4 Serialization time

Our framework will be used when a user wants to migrate a web app's state, so the time it takes to serialize may degrade the user experience. To check the usability of our framework, we measured and evaluated the serialization time. The results on x86 and ARM board were similar in its behavior (different in scale), so we show the graphs on x86 only for simplicity. We broke down the total time to four categories:

1. Traverse Window Property Tree: Time taken to traverse the window property tree and to build a scope tree

2. Function Reordering: Time taken to arrange the scope nodes in the scope tree

3. Code Generation: Time taken to generate restoration code

4. Others: Initialization, DOM-related serialization time

Figure 11a shows the overall results on web apps on x86. The time is mostly spent on traversing the window property tree. Function reordering and code generation are finished in relatively shorter time. In percentage, Function reordering and code generation took 6.7% to 13.9% of the total serialization time, which is acceptable. Also, the total serialization time is at most 123ms ('Phrase'), which would be instantaneous to a user. It took a little longer time to generate code in 'Phrase' because it took extra time to generate huge restoration code for the image data. On ARM board, the serialization time took around 10 20 times to that of x86 (927 2105ms). Even in the worst case (2105ms for 'Phrase'), we believe that a user can wait 2 seconds if he/she can migrate and continue the web app in other devices.

Figure 11b shows the overall results on Octane benchmarks on x86. The benchmarks that took relatively short time to serialize (code-load, crypto, deltablue, navier-stokes,

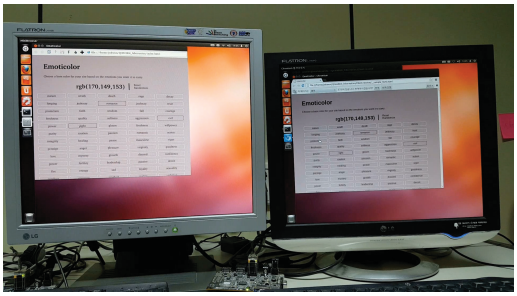

Figure 13: Web app migration demo (Emoticolor)

raytrace, regexp, Richards, splay, zlib) are benchmarks that have relatively small source code size (Table 1) and have fewer JavaScript variables. 'pdfjs' generates an object that has numerous properties, and each property has its own numerous properties, so traversing and code generation take a large amount of time. 'mandreel' is a benchmark that does not create any LEs, but it generates numerous function objects that belong to the global scope. Therefore, function reordering is needed to be done on many functions, resulting in a longer time taken in the traversing step and function reordering step.

6.5 Limitation test

We created two microbenchmarks to test our framework with some extreme situations. The first experiment is done to test how many scope tree heights that our framework can support. We wrote a microbenchmark that creates a function that has scope tree height of n. We varied the input n from 1 to 2500 to see the change in serialization time. The result is shown in Figure 12. When the number of scope tree height is increased, the code generation time is remarkably increased. It is because we need to do the recursive call n times during code generation time, which becomes an overhead. On the other hand, we wrote another microbenchmark that creates n scope nodes that share the same parent scope node. We again varied the input n from 1 to 2500. In this case (Figure 12b), the code generation time remained short but function reordering time had increased. It is because function reordering time depends highly on the number of scope nodes that are at the same level. In this case, we have to check n number of scope nodes from each other to arrange the initialization order, so it takes longer time.

6.6 Demo

Figure 13 shows the working demo of the migration of a web app (Emoticolor). Emoticolor exploits jQuery library, so closures are heavily used. The migration is done between two embedded devices (left to right), showing that our framework is applicable to ubiquitous and mobile computing. The full video can be accessed via `https://youtu.be/_qgxSsvDkZs`.

7. RELATED WORKS

In distributed systems, *mobile agent* [4] is suggested to distribute workloads to multiple nodes. The concept of mobile agent is implemented in many languages, such as C++ [3] and JAVA [1]. A system that deploys a mobile agent can migrate an agent containing its data states, executable codes and possibly execution states. Execution states can be migrated only if the system has implemented strong migration. Mobile agents are similar to our work in some ways that they both migrate the code and the state of a program. However, there are differences between our work and mobile agents. The major difference is that our work is transparent to the developers, meaning that the developers do not need to know at all about our migration system to enable migration, while the developers must study and instrument their codes to enable migration in mobile agents. Also, the migration of mobile agents requires mobile agent platform at the target device, while the migration of our work requires an ordinary browser to run the migrated codes.

In Scheme language, G. Germain et al. [8] constructed a framework that can serialize closure and continuation (control state) of a function. They implemented the idea to exploit distributed computing in Scheme. In their work (named Termite), a Termite process is lightweight and can be spawn in multiple numbers without huge overhead. In such situation, a function can be serialized with its environment and continuation, and it can be executed in other Termite processes. The major difference between Termite and our work is that Termite introduces new grammar that Termite Scheme developers should study, while our work is totally transparent to JavaScript developers so that it can be applied to any web app. Also, Termite Scheme developers need to designate the point when the process migration should be done, but our migration can be done at any time.

8. SUMMARY AND FUTURE WORKS

We have presented a solution to serialize a closure's state completely by reconstructing the environments. We have studied the language semantics and actual implementations in a JavaScript engine to completely check the state information we needed to serialize. We implemented our work on an open source browser, and our work could serialize and restore seven real closure-heavy applications and full Octane benchmark set within acceptable overheads.

We have implemented our work based on the ECMAScript 5.1 specification, and recently (June 2016) ECMAScript 7 [7] is accepted as a standard. There are new features that are added in ECMAScript 7 that our work may not support. For example, 'let' statement is introduced in ECMAScript 6, and it allows a declaration of a variable in a specific block. We expect that the support for 'let' statement can be simply done in our work since it does not break the semantics of the LEs, but the implementation is left as a future work.

Furthermore, we look forward to supporting serialization of other parts of web apps that are currently not supported. One example is to serialize CSS animation's states. Many recent web apps exploit CSS animation feature to add some dynamic behaviors. Currently, there is no way to serialize the state of CSS animation, and we expect that restoring CSS animation state will be even more challenging since we have to restore the exact position and shape of the current state. This is left as a future work.

9. ACKNOWLEDGMENTS

This work was supported by the Ministry of Trade, Industry & Energy (MOTIE) through the Electronics and Telecommunications Research Institute (ETRI). (Project No. 1004 5344)

10. REFERENCES

[1] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[2] F. Bellucci, G. Ghiani, F. Paterno, and C. Santoro. Engineering javascript state persistence of web applications migrating across multiple devices. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering Interactive Computing Systems*, pages 105–110, New York, USA, June 2011. ACM.

[3] B. Chen, H. H. Cheng, and J. Palen. Mobile-c: A mobile agent platform for mobile c-c++ agents. *Softw. Pract. Exper.*, 36(15):1711–1733, Dec. 2006.

[4] R. S. Chowhan and R. Purohit. Study of mobile agent server architectures for homogeneous and heterogeneous distributed systems. *International Journal of Computer Applications*, 156(4):32–37, Dec 2016.

[5] Chrome v8. `https://developers.google.com/v8/`.

[6] Ecmascript language specification, standard ecma-262, 5.1 edition. `http://www.ecma-international.org/ecma-262/5.1/`.

[7] Ecmascript language specification, standard ecma-262, 7th edition. `http://www.ecma-international.org/ecma-262/7.0/`.

[8] G. Germain, M. Feeley, and S. Monnier. Concurrency oriented programming in termite scheme. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 20–20, New York, USA, September 2006. ACM.

[9] Language trends on github. `https://github.com/blog/2047-language-trends-on-github`.

[10] jquery. `http://www.jquery.com/`.

[11] J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22nd international conference on World Wide Web*, pages 815–826, New York, USA, May 2013. ACM.

[12] Octane: The javascript benchmark suite for the modern web. `https://developers.google.com/octane/`.

[13] J. Oh, J. Kwon, H. Park, and S. Moon. Migration of web applications with seamless execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 173–185, New York, USA, March 2015. ACM.

[14] prototype. `http://prototypejs.org/`.

[15] Webkit: Open source web browser engine. `https://webkit.org/`.