

JavaScript Multithread Framework for Asynchronous Processing *

Daisuke Maki

Graduate School of Electro-Communications
The University of Electro-Communications
maki@ipl.cs.uec.ac.jp

Hideya Iwasaki

Department of Computer Science
The University of Electro-Communications
iwasaki@cs.uec.ac.jp

ABSTRACT

Although Ajax is widely used in the development of Web applications, it is well known that Ajax development is much more difficult than traditional Web development. There are two reasons: (1) Ajax developers have to write complex asynchronous program on a single thread; (2) asynchronous communication on JavaScript can be programmed only in event driven style, which causes control-flow difficulty. To resolve this problem, we provide multithread library to JavaScript programmers. The proposed library has the following features: (1) it is portable among popular Web browsers; (2) it provides preemptive scheduling; (3) it provides object-oriented API. The proposed system converts JavaScript programs written in multithreaded-style into those in continuation-based style that are executable on existing systems, and then executes them concurrently on a runtime-library called thread-scheduler. To see the effectiveness of the library, we implemented an Ajax application using the library. The overhead of the converted programs is not a serious problem in practice because the overhead is smaller enough than communication delay of Ajax applications.

1. INTRODUCTION

Though Ajax applications such as Google Maps have become widely recognized, it is also well known that Ajax application development is extremely difficult. There are two main reasons for this difficulty: (1) applications that require complex user interaction must be developed in a single-threaded execution environment, and (2) asynchronous communication in JavaScript can be programmed only in event driven style, which causes control flow difficulty. The JavaScript thread model, which is a preemptive single-thread environment, creates both of these difficulties.

*This work is partially assisted by Exploratory Software Project 2006 from IPA (Information-technology Promotion Agency, Japan).

In order to solve this problem, in this research we propose the use of multithreaded programming to JavaScript programmers, and have provided a library that provides a multithreaded environment in JavaScript. This library has the following characteristics.

- Written in JavaScript, and no modifications have been made to the JavaScript implementation
- Portable among popular Web browsers
- Provides preemptive scheduling
- Provides object-oriented API

The widespread popularity of Ajax is due to the increased compatibility between Web browsers, which has made it possible for Ajax applications to be executed on a variety of Web browsers. Considering this, our proposed system design places importance on portability among Web browsers to ensure that the system can be used in actual Ajax development. As a result, instead of relying on specialized systems, we use code conversion to make the library available on existing systems. The code conversion involves rewriting JavaScript programs written in multithreaded style into regular JavaScript programs that are executable on existing systems. This rewrite uses continuation-based concurrency[11, 14]. More specifically, the operation involves dividing the program into small sections, and inserting code in between that performs a type of context switch. At that time, in order to explicitly preserve control flow, it uses continuation-passing style[1]. It also uses trampolined style[4] to achieve preemptive scheduling. In this way the converted program behaves as if it is running concurrently on multiple threads. In addition, the converted program is executable on existing JavaScript systems because it is compliant with standard JavaScript specifications[3]. Thus, the program is not dependent on a specific Web browser and can achieve portability.

This paper is organized as follows. In Section 2, while simply explaining Ajax and its core technology JavaScript, we will clarify the factors contributing to the difficulty of Ajax application development. In Section 3, we discuss the design objectives and functions that the proposed system, the multithreaded library, should achieve. We also give an overview of the proposed system based on the assumed application usage. Section 4 will explain how those objectives will be

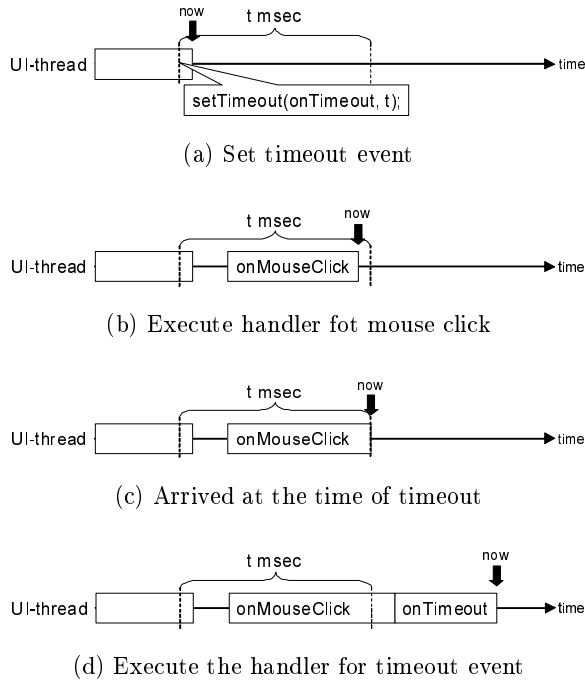


Figure 1: Image of runtime UI-thread

achieved, and the method in which the proposed system can be implemented. In Section 5, we will actually write an Ajax application with the proposed system prototype and discuss its effectiveness. Section 6 will introduce existing research that our proposed method references, as well as related research from other fields. Finally, Section 7 includes a summary and discusses future challenges.

2. JAVASCRIPT AND AJAX

2.1 JavaScript

JavaScript is an object-oriented programming language developed to write dynamic contents within a Webpage. Usually JavaScript refers to the scripting language included in Netscape Navigator, but this paper uses the term to refer to programming languages included in Web browsers that adhere to the ECMAScript 3rd edition[3] standard.

The Web browser performs all JavaScript processing on a single thread. This is the thread that builds the Web browsers user interface, and so is called the UI-thread. Normally JavaScript programs are called as event handlers when mouse clicks or other events occur. Each individual event handler is then executed non-preemptively on the UI-thread. The system is such that while one event handler is being executed, no other events can be executed at the same time, nor can they interrupt the event handler currently being executed.

For example, consider the condition of the UI-thread when a JavaScript program is executed, in which a timeout event and mouse event occur. Figure 1(a) shows the state in which the program timeout event has been scheduled, and the program has ended. In JavaScript, the built-in `setTimeout` function is used to set the timeout. This function takes

a function to be called as an event handler as the first argument, and the time in milliseconds until the timeout will occur as the second argument. The function then returns the integer value representing the scheduled timer identifier. Here, before the timeout occurs, the user's mouse-click causes the `onClick` event handler to be executed and the condition becomes as shown in Figure 1(b). As shown in Figure 1(c), until this `onClick` is finished processing, even if the time comes for the previously scheduled timeout to execute, the event will not occur at the time when it was originally intended. This is because, as previously mentioned, while one event handler is being executed, no other events can be executed. The actual time the timeout event will be executed, is after the `onClick` processing has been completed, as shown in Figure 1(d). This example illustrates that JavaScript events are registered in a queue internal to the Web browser, and then, when the UI-thread is free, executable events are taken and executed sequentially from the queues.

The constraint, in which while one event handler is being executed no other events can be executed or interrupt, also applies to the user interface drawing. Therefore, when processing that takes a long time occupies the UI-thread for an extended period of time, to the user it appears that the Web browser has frozen. A typical example is an infinite loop such as the one shown below.

```
while ( 1 ) do_stuff();
```

When this type of program is executed, the UI-thread will not be freed, and the Web browser will freeze. Therefore, when writing JavaScript programs, one must be sure not to create a situation in which the UI-thread will be occupied for an extended period of time.

JavaScript does not have a command capable of suspending or restarting the current processing. Thus, the only way to free an occupied UI-thread is to return from the event handler that is being executed, and completely stop processing entirely. Also, because the event handler constantly calls for the restart of the execution, execution will start from the beginning of the function. For this reason, it is difficult to create JavaScript programs that execute while appropriately releasing UI-threads. In systems that actually require the release of UI-threads, unfortunately loop statements, function calls, or other control flow statements cannot be used, which forces the programmer to write extremely complex code. This control-flow problem will be explained in detail in Section 2.2.2.

2.2 Ajax

Ajax (Asynchronous JavaScript + XML)[5] applications, are Web applications that communicate with the server asynchronously.

Traditional Web applications were basically only able to communicate with the server at the time when the page was being loaded. If it were necessary to transfer results of user input to the server, the entire page had to be read again from scratch by accessing a page, or reloading the current

```

1: var req = new XMLHttpRequest();
2: req.open("GET", url, true);
3: req.onreadystatechange = function () {
4:   if (req.readyState == 4) {
5:     if(req.status == 200)
6:       document.write(req.responseText);
7:     else
8:       document.write("ERROR");
9:   }
10: };
11: req.send(null);

```

Figure 2: Example of use of XMLHttpRequest

page. With this method, the only thing the user could do while the webpage was being loaded was wait. Therefore applications that required frequent data transfer experienced extremely poor user response.

On the other hand, Ajax applications communicate with the Web server from JavaScript programs. By doing this, it is possible to send or receive new data without reloading the webpage, and thus create Web applications with excellent user response.

2.2.1 Server-communication on JavaScript

In order to communicate from JavaScript to the Web server, the XMLHttpRequest object is used. The basic usage of XMLHttpRequest is shown in Figure 2.

The meanings of the arguments in the open method on the second line are as follows: the HTTP request method, URL, and the true or false Boolean value that indicates whether or not asynchronous communications will be used. The third argument is essentially always set to true. The reason that the third argument is always set to true is because, as mentioned in section 2.1, the UI-thread is occupied while JavaScript is being executed. Therefore, while waiting for a server response in synchronous communications, it will appear that the Web browser has frozen. Applications of this nature are undesirable in terms of usability, so only asynchronous communications can be used.

On line 3, a function has been assigned to the onreadystatechange property. This registers the event handler for the XMLHttpRequest object's readystatechange event. This event will be executed every time the object's state changes during asynchronous communications. We can know the object's current state via the readyState property, and the values and the corresponding states are assigned as shown below[9].

```

0: Uninitialized
1: Open
2: Sent
3: Receiving
4: Loaded

```

Finally, as can be seen in line 11, the request is sent to the server via the send method. The argument for send is the

HTTP request body, but in the HTTP GET method, the request body does not exist, so in this case the value that is passed is null. Afterwards, every time the object state changes, the event handler function assigned to onreadystatechange will be called. The program described will wait until loading is complete and the object status becomes 4 (Loaded). After loading is finished, the response status value will be checked. If a valid response is confirmed, the loaded contents will be displayed. Otherwise a character string indicating an error will be displayed.

2.2.2 Ajax Programming Problems

As shown in Figure 2, the asynchronous communications used in Ajax applications are written in an event driven style. This means that in order to get the communication results, the event handler being executed must first be stopped. Practical application programs must communicate multiple times to complete a single job, so using this method could cause the program to be divided at each communication.

For example, consider a program that gets data from the server, processes it in some way, and then returns data to the server. This is processed sequentially, but when implemented we get the code shown in Figure 3(a). Looking at this code, it is difficult to understand the sequence that the code will be executed, or what the program actually does. Actually, lines 1 to 4 will be executed, and then the program will end, freeing the UI-thread. Event notifications will cause handler1 to be executed, and then the program will end again at line 13. Finally, handler2 at line 20 will be executed.

However, writing the same program with synchronous communications allows it to be written much more simply, as shown in Figure 3(b). Also, the program written this way is much easier to understand. This is because the program is not being divided at each communication, and it can directly benefit from the constructs provided by the JavaScript language. However, as described in section 2.1, because of the restrictions in JavaScript's thread model, the program cannot be written this way in actual Ajax development.

Abstraction with functions, which is the common way to achieve modularity, is difficult in Ajax development, and is one of the problems that Ajax faces. For instance, consider user authentication procedure authenticate, using Ajax. This can be abstracted as a function that takes in a username and password, and returns true if they are the right combination, and false if they are not. However, if the authentication processing happens on the server-side, this type of interface becomes inconvenient. The fact that communication with the server is necessary in the authentication process means that event-driven asynchronous communication must be used. In other words, the program must be stopped in order to receive the authentication result from the server, so the result cannot be returned as the return value of the authenticate function. Instead, the function callback must be specified, which is called and passed the authentication result after the asynchronous communication has completed. Therefore, the authenticate interface must be written as follows.

authenticate (name, pass, callback)

Through this example it is apparent that when changes in the internal implementation can affect the interface visible from the outside, it causes a problem with modularity.

Both of these problems are caused by the division of the program at each communication. The necessity to end the program with each communication means that many language features of JavaScript, such as loop statements, try-catch statements, other constructs, and function calls, cannot be used to achieve flow control. This control-flow problem complicates Ajax development. This is because communications are event-driven in JavaScript, and therefore the source of the problem is in JavaScript's thread model. The important point is that as long as Ajax programming depends on event driven programming, this issue cannot be avoided. No matter how the primitives for the event driven asynchronous communications is skillfully wrapped, it will not solve this essential problem.

3. PROPOSED METHOD

3.1 Multithreaded Programming

As noted in the previous section, as long as event driven programming with asynchronous communications is used, it is not possible to avoid Ajax programming complexity. The reason for this is that all asynchronous processing must be done on a single thread. If multiple threads were available to allow processing while waiting for server response, it would be possible to create programs without losing either descriptivity or user responsiveness.

In this research we propose a method to achieve asynchronous processing through multithreaded programming, instead of through event driven programming, in order to eliminate the complexity of Ajax programming. For this, we aim to provide a multithreaded programming environment for JavaScript.

3.2 Design

The basic direction taken to provide a multithreaded programming environment was to provide the proposed system as a JavaScript library. In other words, nothing was modified regarding the Web browser, and a goal was set to maintain portability among the popular Web browsers. The reason for this decision was that one of the reasons that the target for this research, Ajax, gained popularity was that compatibility among Web browsers increased. Even if we were to succeed in making it possible to utilize multiple threads on one particular Web browser, it would not be a significant achievement because it would be useless in actual Ajax application development.

In order to create the library, the basic approach being considered is timesharing the UI-threads to provide multithreads. The planned features include preemptive scheduling and an object-oriented API. The proposed system has implemented code conversion in order to achieve multithreading, the details of which will be explained in detail in section 4. JavaScript evaluates code generated at run-time, and code obtained from the server with asynchronous communications via the built-in eval function. Therefore, in order

```
1: var req = new XMLHttpRequest();
2: req.open("GET", url, true);
3: req.onreadystatechange=handler1;
4: req.send(null);
5:
6: function handler1 () {
7:   if (req.readyState == 4) {
8:     if (req.status == 200) {
9:       var text = req.responseText;
10:      Modify text
11:      req.open("POST", url, true);
12:      req.onreadystatechange=handler2;
13:      req.send(text);
14:    } else {
15:      onError("ERROR: can't GET");
16:    }
17:  }
18: }
19:
20: function handler2 () {
21:   if (req.readyState == 4) {
22:     if (req.status == 200) {
23:       document.write("OK");
24:     } else {
25:       onError("ERROR: can't POST");
26:     }
27:   }
28: }
29:
30: function onError () {
31:   document.write("ERROR");
32: }
```

(a) Implementation with asynchronous communication

```
try {
  var req = new XMLHttpRequest();
  req.open("GET", url, false);
  req.send(null);
  if (req.status != 200) throw "ERROR";
  Modify text
  req.open("POST", url, false);
  req.send(text);
  if (req.status != 200) throw "ERROR";
  document.write("OK");
} catch (e) {
  document.write(e);
}
```

(b) Implementation with synchronous communication

Figure 3: Program that communicates with a server

for the code obtained at run-time to be executed on multiple threads as well, the system must be designed such that the code is converted at runtime.

In JavaScript, functions are expressed as Function Objects, and just like all other objects, the Function Object has a toString method. The default behavior as defined in the standard specifications³ of a function's toString method returns a single string representing the source code of the function. This means that at run-time a JavaScript program can obtain its own source code as a character string, using the function as a unit. In this way, the code conversion can also be done using the function as a unit.

The code conversion at run-time will be done only with the functions passed as arguments to the create method, or compile method, which will be explained in the next section. This process will not have a cascading effect, meaning that functions that are called by those functions having their code converted will not also be converted. The foremost reason for this is that, functions provided as built-in functions (for example, input output functions) cannot be written to in JavaScript, so the code for these functions cannot be converted anyway. The second reason is that, as will be explained in detail in section 5, the code conversion in the proposed system has a large overhead; by converting only the necessary code the overhead created for the entire program can be reduced.

The necessary code conversions can be done not only at run-time, but before run-time as well. The processing of the code conversion before run-time are exactly the same as the code conversion required during run-time, so if the sections that require conversions are known in advance, making the conversions before run-time can save CPU time.

3.3 API

The APIs provided by the proposed system as a library are shown in Table 1. These APIs are all defined as properties of the Thread constructor. Thread.Http.get and Thread.Http.post are communication APIs that will not block the UI-thread, and are used in place of the XMLHttpRequest that was described in section 2.2.1.

Next, the API and the functions it provides will be explained while looking at the program written with the proposed library. The program in Figure 4 will be used as an example of the proposed system. This is a simple program to retrieve data from a specified URL and display the contents. To show that the Web browser is not frozen while communicating with the server, a simple animation will be shown in the status bar.

This program defines two functions. The first function, load, retrieves and displays data from the server. The other function, nowLoading, displays the animation. Each function is running on separate threads.

Line 22 is the section in which the load function is called in order to be executed on a separate thread. Thread.create is the method that converts the code of the function specified in the first argument, and executes the modified function on the newly created thread. The remaining arguments

```

1: function load (url) {
2:   var th = Thread.create(nowLoading);
3:   try {
4:     var res = Thread.Http.get(url);
5:     document.write(res.responseText);
6:   } catch (e) {
7:     document.write("ERROR: " + e);
8:   }
9:   th.kill();
10: }
11:
12: function nowLoading () {
13:   var bar = ["|", "/", "-", "\\"];
14:   var i = 0;
15:   while ( true ) {
16:     window.status = "Now loading..."
17:                   + bar[i=(i+1)%4];
18:     Thread.sleep(125);
19:   }
20: }
21:
22: Thread.create(load, "http://...");

```

Figure 4: Example of use of the proposed system

are passed unaltered to the converted function. First, load uses Thread.create, and nowLoading is called on a different thread (line 2). This allows load and nowLoading to operate concurrently.

The return value of Thread.create is a Thread object, which represents the newly created thread. From this point, all operations to be performed on the thread will be performed through this Thread object. For example, the method notify causes an exception on an arbitrary thread. It is possible to catch and process the exception in the thread where it occurred, but if it is not caught the thread execution will be aborted. The kill method in line 9 is a shortened form of the following code.

```
th.notify(new Thread.KillException());
```

The Thread.Http.get on line 4 gets data from the server, but the process will be returned from the Thread.Http.get function after all communication has been completed. It appears that during communications this thread's processing has temporarily frozen, but during this time another thread can continue running. For this reason, there is no need to worry about user response suffering. Also, if any exceptions occur during communication (communication error, etc) they will be caught in the catch section on line 6. In this way, we can write control-flow using the basic constructs provided by JavaScript and alleviate the complexity introduced in section 2.2.2.

The nowLoading function basically contains an infinite while-loop which draws an animation. This function cannot end processing itself, so it waits to be ended by an exception from the notify method called in another thread.

Table 1: List of public methods

static methods	
compile(f)	code-converts function f
create(f, a ₁ , a ₂ , ...)	executes function f on new thread
self()	returns the reference to the current thread
sleep(t)	suspends the current thread in t milliseconds
stop()	stops the current thread unlimittedly
yield()	switches the execution thread
Http.get(u)	retrieves contents of URL u via HTTP GET
Http.post(u, b)	submits string b to URL u via HTTP POST, then returns the response body
instance methods	
join()	stops the current thread until the target thread completes its work
kill()	finishes the target thread
notify(e)	issues an exception e on the target thread

In order to open the interval between animation repaints, the `Thread.sleep` method is used in line 18. This method causes the current thread to temporarily sleep. The length of the pause is measured in milliseconds. While the thread is paused, the `notify` method can be used to cause an exception. In this case execution is restarted immediately without waiting for the timeout to occur.

Another method similar to `Thread.sleep` is also available, it is called `Thread.stop`. This thread causes execution of the current thread to stop, however in this case execution will not be restarted by the timeout, and the thread can only wait for an exception to occur. Therefore, we can assume usage similar to that shown below.

```
try {
  Thread.stop();
} catch (e) {
  execution restarts here
}
```

At the moment, the proposed system does not provide any exclusive control functions. The reasoning for this was as follows. Considering that in most Ajax applications JavaScript is only responsible for the user interface, and most processing that would require exclusive control are executed on the server. It seemed extremely rare that the absence of exclusive control functions would cause a serious problem. Therefore, if the programmer finds it necessary, exclusive control has to be implemented at the user level. However, availability of exclusive access control is vital for this library, so including it is an ongoing challenge.

4. IMPLEMENTATION

4.1 Overview and Configuration

The proposed implementation method uses timesharing to construct a pseudo-multithreaded environment on top of the UI-thread. The basic idea is that the JavaScript program is broken up into its basic blocks and executed, then at the appropriate timing between basic blocks, the UI-thread is released. To achieve this we use continuation-based multithreading, which is used throughout the functional programming language world. Continuation1 is a function that

```
function f ( ) {
  var c = getc();
  // (*)
  document.write(c, "\n");
}
```

(a) Original program

```
function f ( ) {
  var c = getc();
  function rest ( ) {
    document.write(c, "\n");
  }
  rest();
}
```

(b) Rewriting result

Figure 5: Rewriting with continuation

represents "the rest of the computation". It represents the tasks that should be executed after some point in the program in a form of a function. To give a very simple example, using continuation it is possible to rewrite the program in Figure 5(a) as Figure 5(b) without changing the meaning¹.

The continuation that occurs at the asterisk (*) in Figure 5(a), corresponds with the `rest` function after the program rewrite. Assume that we want to suspend execution at the (*), and resume at a later time. Then, we can do so using the string `rest`, and calling it when we want to resume. In other words, storing the thread context is equivalent to storing the continuation, and switching the context is equivalent to calling the stored continuation. Now, two tasks remain; one is to think of an appropriate way to return the continuation (`rest` in this case) to the caller, and store it. The other is to think of an appropriate method and timing to call the stored continuation. This proposed method uses mainly a code

¹Figure 5 is an image to show an example of the code rewrite. The actual rewrite is more complex, and will be explained in future sections.

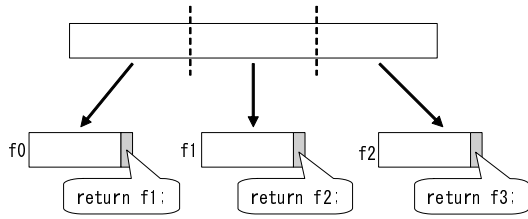


Figure 6: Image of code conversion

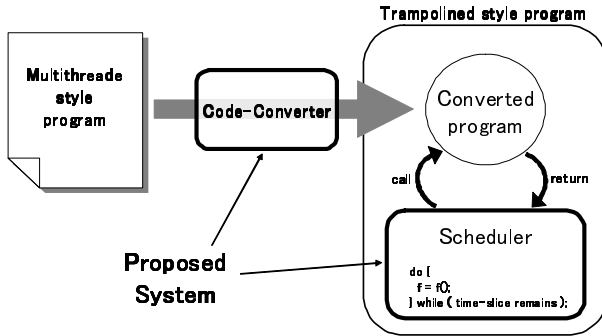


Figure 7: System overview

converter to achieve the former, and a thread scheduler to achieve the latter. By using code conversion, we achieved the goal of making multithreaded programming possible without modifying the JavaScript implementation and maintained compatibility with existing systems.

The code converter takes a regular JavaScript program and breaks it into multiple functions by its basic blocks, and each function is rewritten so that it returns continuation after processing progresses for some time. In other words, the program is divided up into individual functions as shown in Figure 6, and code to return the function to be executed next (actually the context object, to be further explained in section 4.2) is inserted at the end of each function (the boundaries of the divisions). Then, each of the divided functions is called by the scheduler. The scheduler calls the divided function, then the returned function is called again, and this process is repeated. When a context switch is needed, such as for preemption, an event handler is registered to call the returned function later, and the UI-thread is released. As previously mentioned, by proceeding with the execution of these basic blocks, the UI-thread can be released at the appropriate timing between basic blocks. The code converter changes the program into the form which the scheduler expects without changing its purpose. The overview of the proposed system is shown in Figure 7.

4.2 Context Objects and Continuation Objects

When considering what the scheduler should take in as context of the program being executed, allowing the divided functions to simply return the next function to be executed, the design goal set in section 3.2 cannot be achieved. In addition to this, a means to inform of the thread suspension, such as the sleep method, and a value to pass as an argu-

ment when the next continuation is called are needed. For this, each of the functions always returns an object with the following 3 properties.

```
{
  continuation: continuation,
  timeout      : time to wait until restarting,
  ret_val      : actual arguments on continuation
}
```

Hereafter, this object will be referred to as the "context object". The timeout property indicates the time to stop in milliseconds, but when the program cannot be restarted based on time, it is indicated with a negative number. Also, a fixed period of time known as a "time slice" is introduced. If it is within the time slice and it is not necessary to switch the thread, processing can continue without releasing the UI-thread. Whether or not it is necessary to switch the thread is indicated by setting the timeout property undefined value.

However, there are still a few points that make it insufficient to represent continuation, the "rest of the computation", with the use of functions only in JavaScript. Those are the following two points.

- The value of this
- Exception handler

In JavaScript, this value for the function (method) code execution is determined with each function call. For this reason the value that this represents must be remembered each time a function that represents continuation is called. Also, in addition to regular continuation, the continuation for exception occurrences must also be remembered. As explained in section 3.3, by using the notify method, exceptions can be issued externally to the thread. This makes it necessary to give the thread that caused the exception access to the exception handler..

With the above discussion, we can express continuation in JavaScript with an object that has the following properties.

```
{
  procedure: function to represent the rest of the computation,
  this_val : this value in the function,
  exception: continuation when an exception occurs
}
```

The above is called the "continuation object". The procedure property is a 1 argument function, and when it is called the context object must be returned. The exception property points to the continuation object, so the chain of the continuation objects from the exception property takes the form of a list. From the run-time point of view, this expresses the nested structure of the try-catch, which has dynamic scope. Therefore, the relationship between the continuation object and the context object are as shown in Figure 8.

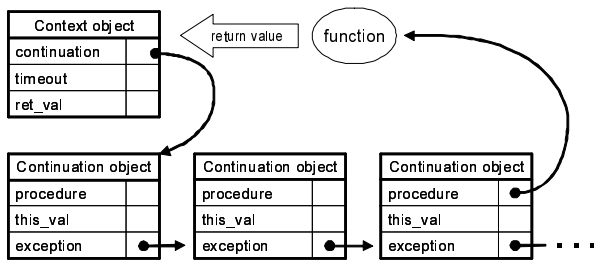


Figure 8: Relationship of objects

4.3 Scheduler

4.3.1 Trampoline usage

Because the main task of the scheduler is to get the context object and repeatedly call the continuation stored inside, the processing centers around a loop such as the one shown below.

```
do {
  try {
    context =
      context.continuation.procedure.call(
        context.continuation.this_val,
        context.ret_val
      );
  } catch (e) {
    exception handling
  }
} while (context.timeout == undefined
  && time slice remains);
```

This is known as a trampoline^[4]², and is a method commonly used in compiler implementation^[6, 7, 13, 15]. After this trampoline has been exited for the context switch and continuation remains that needs to be called, the thread execution has not yet completed so the context must be stored to restart the thread later. Since the thread context is packed in the context object, the context object is available to preserve the context. The problem is how to schedule the trampoline to be executed again later. When doing a context switch, it is necessary to completely end the event that executed the trampoline and release the UI-thread. This makes it impossible to simply put the trampoline inside of a loop statement and construct it to run in a round-robin fashion.

The proposed system uses the built-in JavaScript function `setTimeout` to restart execution of the scheduler. The `setTimeout` function can generate an event without user interaction, so it is suited for this situation. As explained in section 2.1, events created by `setTimeout` are stored in the browser's internal event queue. Each time the UI-thread is released, the single event (only applies to events that are able to be issued) in the front of the queue will be issued (as a result, the event handler for the event that was issued is executed on the UI-thread), so this corresponds to the scheduling activity. Thus the actual schedule processing

²Also referred to as a Dispatch Loop^[13], or UWO handler^[6].

is completely left up to the internal processing of the `setTimeout` function by registering one event per thread in the event queue. Therefore, a Thread object is a data structure as shown below, consisting of an integer that identifies the timer (which is the `setTimeout` function return value), and a context object.

```
{
  timerID: timer identifier,
  context: context object
}
```

The event scheduling is done with the thread object's private method `standBy`.

```
function standBy ( t ) {
  // if the event is already registered in the queue
  // it is cleared to avoid duplication
  if (this.timerID !== undefined)
    clearTimeout(this.timerID);
  var self = this;
  this.timerID = setTimeout(
    function(){ self.doNext(); },
    t
  );
}
```

Here, `doNext` is the method that implements the trampoline. So the thread for the current implementation is a combination of the context object which stores the rest of the computation for the thread, and the timer which schedules the threads restart event. When the timer alerts of a timeout event, the event handler uses the corresponding context object to start the trampoline loop.

For example, when preemption occurs due to a time slice expiring, after exiting the trampoline loop `standBy(0)` will be executed. Here, the real argument for `standBy` is 0, which represents 0 milliseconds. In other words, it is registered in the event queue so that the timer event will occur as quickly as possible. As explained previously, the events in the queue can only be fetched one by one sequentially as the UI-thread becomes free, so in this way scheduling can be achieved in a round robin fashion.

An execution image of the UI-thread is shown in Figure 9. As shown in the figure, each time a time slice expires, the trampoline code explicitly releases the UI-thread. This allows mouse clicks and other event handlers to be executed in the gap between time slices, and the program can respond to the user without any large delay.

4.3.2 Sleep and Exception Occurrences

The sleep method, which makes the current thread sleep, is defined as a method that simply returns a context object with its `timeout` property set with the given argument, the sleep time. It is as follows.

```
function sleep ( $this, $args, $cont ) {
  return {continuation: $cont,
```

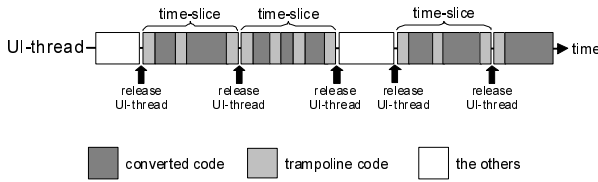



Figure 9: Image of UI-thread when using provided system

```

ret_val    : undefined,
timeout    : $args[0] };
}

```

When the scheduler receives the return value from this function, the trampoline loop ends and the `standBy` function is called with the sleep time set in the `timeout` property as its real argument. In this way, we are able to make the thread sleep. The special arguments *contandargs* introduced here will be explained in section 4.4.

The `notify` method, that causes exceptions in another thread, also uses `standBy`. The `notify` method needs to restart the thread if it is sleeping due to a sleep or stop method, so the corresponding timer must be canceled, and the continuation property of the context object that the thread is storing must be set as the continuation for the time the exception occurred. Then the `standBy` is called with 0 as an argument.

```

function notify ( e ) {
  this.context.continuation
    = this.context.continuation.exception;
  this.context.ret_val = e;
  this.standBy(0);
}

```

Now an event is scheduled to restart the execution of the target thread (this) from the continuation corresponding to the exception handler (catch clause). The exception value is passed, via the `ret_val` property, as an argument for the function that the procedure property of the continuation object points to.

By proactively utilizing the event processing system provided by the underlying implementation, the issue regarding the timing and scheduling of when to restart the thread can all be input to the underlying event system via `timeout` event. In addition, because the underlying event system is being used, it also has the benefit that other user interactions during execution, such as mouse events or key events, can be scheduled within the same framework. This type of implementation has the benefit of being concise, and making use of the underlying features, but further consideration will be needed if thread prioritization, or other forms of more detailed scheduling becomes necessary in the future.

Once the all the work of a thread has been completed through the processing of repeated continuation calls, a `NoContinuationException` will occur. Therefore, when a situation occurs

where there are no longer computations remaining, a function like the one below is set in the procedure property of the continuation object.

```

function ( r ) {
  throw new NoContinuationException(r);
}

```

In this way, exceptions thrown will be caught in the trampolines catch clause and the threads job will end.

4.4 Code Converter

The code converter is a program that takes the JavaScript program as input, and outputs program source in the form that the scheduler is expecting. As explained in section 3.2, the code conversion operates using a function as a unit. Then, by evaluating the resulting source code with the `eval` function, a function that operates in the way that the scheduler expects can be obtained at run-time.

The converted function must return the context object as the current context according to the expectation of the scheduler. To do this, the function must receive a continuation object when it is called. This is done in order to retain the return point from the function, and this style is called continuation-passing style[1]. Also, the `this` value is explicitly passed as an argument. In order to ensure the added arguments, the `this` value and continuation object, do not mix with existing arguments, the existing arguments are passed separately together in a single array.

A unique problem in JavaScript regarding the code modification is that, by using the `with` statement, the scope of variables can be changed at run-time. This means that the only time when scope can be resolved is at run-time. This gives the variable name an essentially important role. For this reason, even the name of a local variable cannot be changed through the code conversion. To avoid a variable name conflict under this restriction, new variables introduced during the code conversion process have a '\$' character appended to the front of the variable name³. Avoiding a conflict with this type of naming convention will not always work depending on the type of program received as input, but realistically it is an acceptable solution.

The Arguments object is another problem specific to JavaScript. This is an object created when functions are called, and is an array-like object, that stores the number of real arguments and values. Arguments objects can be referenced through variable arguments in function codes, and are used when writing functions with variable arguments. However, because after the code conversion the program is broken into its separate functions by basic blocks, the arguments value will be different for each basic block. Also, as a special characteristic of the Arguments object, if an array element of an Arguments object is updated, the corresponding parameter variable will also be updated. For example, when the following code is executed, "overwritten" will be displayed on the screen.

³In JavaScript variable names can include the '\$' character. However, according to standard specifications[3] this should be used only with mechanically generated variables.

```
function f (x) {
  arguments[0] = "overwritten";
  document.write(x);
}
f("argument");
```

In JavaScript the only concept with this type of special relationship between variables and array elements is the Arguments object. However, as explained previously, after functions have been converted the way they pass arguments changes, so even if substituted for the arguments object array elements, the parameters values will not be updated at the same time, as the user would expect. Since the only time that an Arguments object can be created is at the time of a function call, in order to handle this issue, the built-in method apply must be used inside of the converted function, to apply the function once more and create this special relationship.

According to the discussion above, the converted function will be in the form shown below.

```
function (
  $this, // this value
  $args, // arguments array
  $cont  // continuation object
) {
  return function ( parameter list ) {
    $args = arguments;
    local variable declaration
    continuation object definition
    ...
    return context object;
  }.apply($this, $args);
}
```

The "parameter list" here uses exactly the same parameter list as the function before code conversion. In this way the arguments array (Arguments object) can be created in which the special relationship between the parameter and array elements holds just as expected. By storing the Arguments object created this way in the \$args variable, they become useable in the body of the converted function mentioned below.

The main contents of the converted function is, as shown in Figure 6, a program that has been divided up and represented as a sequence of continuation objects. In order to restore the arguments variable value in the function that the procedure property of each continuation object is pointing to, the general form that the function the procedure property points to takes is as shown below.

```
function ($ret_val) {
  arguments = $args;
  task to process with this continuation
  return context object;
}
```

The continuation property of the context object indicates the next task that should be executed, in other words, the

jump destination. In that sense, there is an analogy between returning a context object and goto statement. Using this, the first step in code conversion is to rewrite the input program to a form using goto statements and labels. After the rewrite, the basic blocks will almost directly correspond to continuation procedures, as will the goto statements to return statements. However, care must be taken so that the continuation always returns a context object.

4.5 Code Conversion Example

An example of the code conversion is shown in Figure 10. First the program in Figure 10(a) is rewritten into a form using goto statements and labels. The result is shown in Figure 10(b). Then, it is further rewritten so that basic blocks are continuation procedures, and so that goto statements will return context objects. Then, by putting it into the form of the converted function determined above, the program will end up as shown in Figure 10(c). Pay special attention that, with the rewrite of basic block \$LABEL0, code has been added to lines 9 through 11 that corresponds with the goto statements so that context objects will be returned after the code conversion.

Up to this point the details regarding the method to rewrite the return statement have not been discussed, but the return value should be passed to the next continuation using the context object ret_val property. Other than that, the only thing that has been done is the replacements thus far.

The conversion of function calls is a little complex. As explained in section 3.2, the proposed system will not convert the functions used by the code to be converted in a cascading fashion. The representative examples of those functions that would not be converted are built-in functions. However, it is nearly impossible to write a practical JavaScript program without using any built-in functions, so it must be possible for converted functions with converted code to call other converted functions, as well as unconverted. To handle this requirement, the resulting code branches on the condition; one case for converted functions, the other for unconverted. For example, refer to the following code fragment:

```
r = o.m(x, y);
```

is converted into the code below.

```
var $LABELa = {
  procedure: function ($ret_val) {
    ...
    if (o.m is converted code) {
      return o.m(o, [x, y], $LABELb);
    } else {
      return { continuation: $LABELb,
        timeout      : undefined,
        ret_val      : o.m(x, y) };
    }
  },
  ...
};
var $LABELb = {
  procedure: function ($ret_val) {
```

```
function fib (n) {
  var x, y, z;
  x = y = 1;
  while ( n != 0 ) {
    z = x + y;
    y = x;
    x = z;
    n--;
  }
  return x;
}
```

(a) Original program

```
function fib ( n ) {
  var x, y, z;
  $LABEL0:
  x = y = 1;
  $LABEL1:
  if ( !(n != 0) ) goto $LABEL2;
  z = x + y;
  y = x;
  x = z;
  n--;
  goto $LABEL1;
  $LABEL2:
  return x;
}
```

(b) Rewritten with goto and label

```
1: function fib ($this, $args, $cont) {
2:   return function (n) {
3:     $args = arguments;
4:     var x, y, z;
5:     var $LABEL0 = {
6:       procedure: function ($ret_val) {
7:         arguments = $args;
8:         x = y = 1;
9:         return {continuation:$LABEL1,
10:           timeout      :undefined,
11:           ret_val      :undefined};
12:       },
13:       this_val : this,
14:       exception: $continuation.exception
15:     };
16:     var $LABEL1 = {
17:       procedure: function ($ret_val) {
18:         arguments = $args;
19:         if ( !(n != 0) )
20:           return {continuation:$LABEL2,
21:             timeout      :undefined,
22:             ret_val      :undefined};
23:         z = x + y;
24:         y = x;
25:         x = z;
26:         n--;
27:         return {continuation: $LABEL1,
28:           timeout      : undefined,
29:           ret_val      : undefined};
30:       },
31:       this_val : this,
32:       exception: $continuation.exception
33:     };
34:     var $LABEL2 = {
35:       procedure: function ($ret_val) {
36:         arguments = $args;
37:         return {continuation:$cont,
38:           timeout      :undefined,
39:           ret_val      :x      };
40:       },
41:       this_val : this,
42:       exception: $continuation.exception
43:     };
44:     return $LABEL0;
45:   }.apply($this, $args);
46: }
```

(c) Final result

Figure 10: Example of code conversion

```

arguments = $args;
r = $ret_val;
...
},
...
};

```

As shown above, in the case of a converted function, call it with passing the value, argument list, and continuation, then, just return the result, because the converted function will return a context object. In the case of an unconverted function, it is called as usual, and the return value is returned as the `ret_val` property of the context object. When a function call is converted, be sure that function calls are not nested inside of its argument list. This is the same for other types of expressions as well.

5. EVALUATION

We implemented a prototype supporting a JavaScript subset language and evaluated the proposed system. Currently, the prototype only supports most of the expressions, if statements, while statements, and try-catch statements, but these are sufficient to write basic programs.

First, as a qualitative evaluation, we implemented a simple Ajax application, and evaluated the descriptivity of the proposed system. Next, a quantitative evaluation was done regarding the overhead of the proposed system.

5.1 Descriptivity

In order to compare with the descriptivity of the traditional method, in a concrete application, the simple tree structured bulletin board system shown in Figure 11 was implemented in Ajax using two different methods and then evaluated. One application was implemented with the proposed system prototype, and the other was implemented with Prototype[12]. Prototype is a widely-used library for Ajax, and can do not only asynchronous communications, but also includes a GUI library and a collection of many others. Prototype's asynchronous communications library is constructed as an XMLHttpRequest wrapper, and uses an event driven interface. For this reason, the problem described in section 2.2.2 will be present.

The tree bulletin board does not read article data all at once. It acquires data via asynchronous communications after receiving a request to display child node articles of each article in a demand driven operation. If there is no request, the data will be preloaded in the background with asynchronous communications.

The actual description of the backgroundLoad procedure to preload data is shown in Figure 12. First the program using the proposed system in Figure 12(b) will be explained. backgroundLoad accepts the IDs of the articles that the user wants to load as an array. For each article, getArticle is applied. The actual communications processing is performed in this getArticle with the Thread.Http.get method. Therefore, each time the data for an article is acquired, one HTTP connection is established. Once backgroundLoad has obtained the article data, backgroundLoad is applied recursively to the child node articles. Because it is tree struc-

Table 2: Comparison of lines and functions of code

	Line of codes	Number of functions
Prototype	156	27
Proposed system	137	15

tured, by traversing the nodes recursively, the entire tree can be read in depth-first order.

On the other hand, the tree bulletin board implemented with Prototype is shown in Figure 12(a). Essentially the approach is the same as when the proposed system was used, however this time the iteration for the array in backgroundLoad are more difficult to comprehend. The implementation creates the loop by calling the function loop with an indirect recursive call. This is because control flow is hand coded without the use of control-flow statements. As a result, the program is lacking in terms of readability and maintainability. This is caused by the fact that the function doing the communications processing, getArticle, is an event driven interface. Also, using this method, if an error in communication occurs, it is difficult to send a notice to the caller for exception handling. Compared to this, when using the proposed system implementation, exceptions are sent to the caller of backgroundLoad as normal. Thus, we see that the implementation using the Prototype has encountered the problems mentioned in section 2.2.2.

Next we will look at the effects of the proposed system quantitatively. The number of lines of code and the number of function definitions of both implementations of the tree bulletin board are shown in Table 2⁴. As explained up to this point, in event driven Ajax programming each event handler is defined as a function, and functions pass to each other to achieve control flow. This causes the total number of functions to increase and complicate the program.

There is a not a large difference in the number of lines of code. This is because the proposed system is not meant to reuse the implementation. In other words, the programmers must write all of the code that they have to implement. Thus, both will be approximately the same number of lines. On the other hand, there is a large difference in the number of defined functions. Compared to the Prototype, it was possible to write the program with the proposed system using about half the number of functions. From these results, it can be seen that the proposed system succeeds in reducing the complexity of the program.

5.2 Overhead

We measured the overhead of the proposed system by executing two types of benchmark programs on three different kinds of Web browsers. One benchmark program - prime, calculates the 3000th prime number, and most of the processing was used in a while loop. The other - fib, calculates the 20th Fibonacci number, and in this case most of the processing was occupied by recursive function calls. The environment used for measurement was a computer with a

⁴Both share the code for the GUI, so it has been removed from the chart.

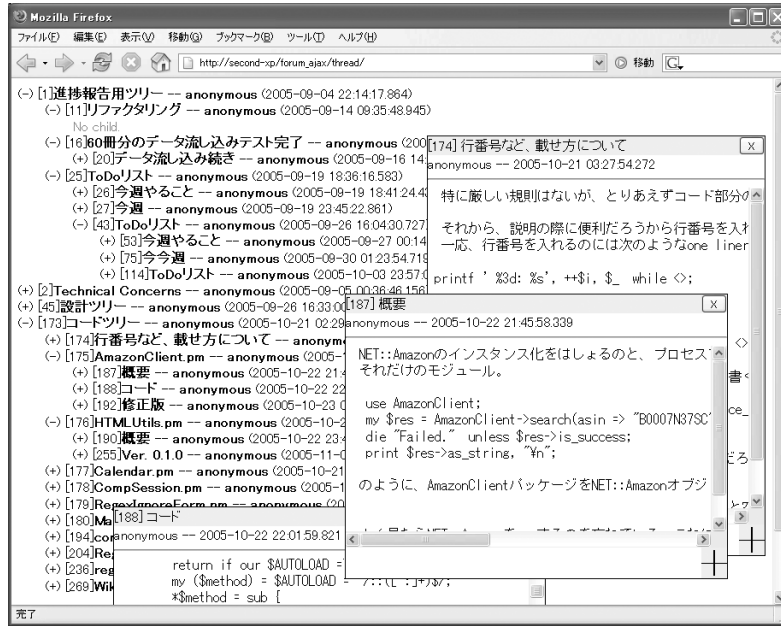


Figure 11: Tree BBS's execution screen

```
function getArticle (id, cont) {
  if (article[id]) {
    cont(article[id]);
  } else {
    new Ajax.Request(
      "../article/?id=" + id, {
        method : "GET",
        onSuccess : function (req) {
          var x, e;
          x = req.responseXML;
          e = x.getElementsByTagName("article");
          cont(new Article(e[0]));
        }
      });
  }
}

function backgroundLoad (ids, cont) {
  var i = 0;
  function loop ( ) {
    if (i < ids.length) {
      getArticle(ids[i++], function(a){
        backgroundLoad(a.children, loop);
      });
    } else {
      cont();
    }
  }
  loop();
}
```

(a) Implemenmtation with Prototype

```
function getArticle (id) {
  if (article[id]) {
    return article[id];
  } else {
    var r, x, e;
    r = Thread.Http.get(
      "../article/?id=" + id
    );
    x = r.responseXML;
    e = x.getElementsByTagName("article");
    return new Article(e[0]);
  }
}

function backgroundLoad (ids) {
  var i = 0;
  while (i < ids.length) {
    var a = getArticle(ids[i++]);
    backgroundLoad(a.children);
  }
}
```

(b) Implementation with the proposed system

Figure 12: Implementations of background preloading

Table 3: Comparison of execution times (in seconds)

	Mozilla FireFox 2.0.0.3		Internet Explorer 7.0.5730.11		Opera 9.20	
	prime	fib	prime	fib	prime	fib
w/ the proposed system	17.72	9.39	10.83	6.53	4.09	1.77
w/o the proposed system	0.06	0.15	0.08	0.09	0.07	0.02
ratio	292	61	144	69	59	98

Table 4: Required time to load articles (in seconds)

	Mozilla Firefox 2.0.0.3	Internet Explorer 7.0.5730.11	Opera 9.20
Proposed system	16.6 ± 1.6	17.7 ± 1.3	14.8 ± 1.3
Prototype	13.4 ± 1.9	16.3 ± 1.5	14.8 ± 1.0
ratio	1.2	1.1	1.0

Pentium 4 3Ghz processor, 1GB RAM, and Windows XP Professional SP2 as the OS.

Table 3 shows the measurement results. Overall, an overhead of 59 to 292 times was observed. Because the results vary depending on the type of browser, generalizations cannot be made, but it can be said that the overhead relating to the while loop is larger. In the program using the proposed system, each code corresponding with a jump in the original source was converted into a return and function call. Measurements showed a great loss of speed due to this point.

The proposed system mainly targets programs that require asynchronous communications, and it is expected that compared to communications delays the overhead will be small enough. This was confirmed by using the tree bulletin board introduced in section 5.1, and measuring the time required to load the article data in the background. The article data used for the evaluation included a total of 303 articles, a total of 807 KB, each article size varied between 355 bytes and 30 KB, and the average size was 2665 bytes. As explained in section 5.1, in order to retrieve one article the tree bulletin board establishes one HTTP connection, so it performed a total of 303 HTTP communications. The client PC was the previously mentioned machine, and the server machine was Pentium M 1.2 Ghz, 512 MB RAM, Windows XP Professional SP2 OS PC. Both machines were connected via 100Mbps Ethernet. The server software used was Apache/2.0.59 (Win32) mod_perl/2.0.3 Perl/v5.8.8. The results of the measurements are shown in chart 4. As indicated by the chart, the difference in the proposed system is hidden in the error and not noticeable. Thus, in practical use the overhead of the proposed system will not be a serious problem.

6. RELATED RESEARCH

Continuation-based concurrency, which this research used, is a well-known method for functional programming languages, especially for languages in which continuation is a first-class object[11, 14]. This is because with those languages continuation of the program being executed can be acquired at an arbitrary point, which makes the context acquisition and storage easy. However in those languages, generally it is not so easy to achieve preemptive scheduling.

Rhino[10] is a JavaScript implementation which places con-

tinuation as its first-class object. However, it is an exception, and at the moment of writing continuation is not a first-class object in JavaScript standard specifications[3]. Also, Rhino is not included in the popular Web browsers. Since this research was focusing mainly on Ajax, Rhino was left out of scope. However, the proposed system uses Rhino code as parts of the code converter by porting it to JavaScript.

In this research, in order to allow JavaScript to handle continuation, we used the method of converting program codes into continuation-passing style (CPS)[1, 2]. In CPS, when function calls are made the function to be executed after that function (continuation) must be passed as an argument. Then, the return from the function is achieved by calling the continuation that was passed as an argument. This is similar to explicitly passing a return address when a subroutine is called in machine language. This allows the programs to achieve control flow while keeping function calls and returns accessible, and have the continuation a first-class object, that can be retrieved or stored as a value.

This type of method that uses code conversion can be regarded as one type of macro-approach. RABBIT[6] compiles Scheme programs into MacLISP programs. Considering it as conversion from a Lisp-dialect to another Lisp-dialect, here, Scheme is taking the role of a type of macro. In RABBIT, call/cc is also implemented with this method. Our proposed method is the same, but it performs conversion from JavaScript into JavaScript.

Li and Zdancewic's work[8] is an example of achieving concurrent processing by code conversion in a programming language other than functional languages. Their research made programming with threads possible on JavaCard, a Java Environment in which threads do not exist. JavaCard applications run on the combination of a host as master, and a card device as slave. The applications are programmed in master-slave programming model using asynchronous communications. Because the asynchronous communication between the master and slave is written in event driven style, the control-flow problem causing Ajax complexity mentioned in this paper also exists in JavaCard programming. Li and Zdancewic planned to solve the problem with the concurrent thread programming model, in which the master and the slave each have a single thread and they perform processing yielding their execution to each other, and convert the

code to conform to the JavaCard spec master-slave style program. In this way, the research done by Li and Zdancewic, and this research have many commonalities. However, while the code conversion for this researched used continuation, Li and Zdancewic used a more ad hoc approach. Another point where the two methods differ greatly is that Li and Zdancewic's solution is constructed in an environment using threads on both the master and slave, while our research aims to construct a multithreaded environment just on the client, the Web browser in this case, to resolve the problem, and the Web server is not involved at all.

7. CONCLUSION

In order to reduce the complexity and difficulty of control flow in Ajax application development, this research proposes writing Ajax applications with multithreaded programming, rather than with event driven programming. Also, to make multithreaded programming possible in JavaScript, a multithreaded library prototype was implemented and its effectiveness confirmed through testing. The proposed system created a multithreaded environment using continuation-based concurrency and code conversion, without altering the JavaScript implementation, and preserving portability.

Consistent effectiveness could be verified using the proposed system by actually implementing an Ajax application, but moving forward it is necessary to accumulate more practical examples and use case results. Also, because the code converted with the proposed system has a large overhead, it is hoped that the implementation can be improved and speed up in the future. Specifically, it is expected that if the conversion result code can be optimized, it will have a great impact on overhead reduction.

8. REFERENCES

- [1] Appel, A. W.: *Compiling with continuations*, Cambridge University Press, New York, NY, USA (1992).
- [2] Danvy, O. and Filinski, A.: *Abstracting control*, LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming, New York, NY, USA, ACM Press, pp. 151–160 (1990).
- [3] ECMA: *ECMAScript Language Specification*, third edition (2000).
- [4] Ganz, S. E., Friedman, D. P. and Wand, M.: *Trampolined style*, ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press, pp. 18–27 (1999).
- [5] Garrett, J. J.: *Ajax: A New Approach to Web Applications* (2005). <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [6] Guy L. Steele, J.: *RABBIT: A Compiler for SCHEME*, Technical Report AI Lab Technical Report AITR-474, MIT AI Lab (1978).
- [7] Jones, S. L. P., Hall, C., Hammond, K. and Partain, W.: *The Glasgow Haskell compiler: a technical overview*, Proceedings of Joint Framework for Information Technology Technical Conference, pp. 249–257 (1993).
- [8] Li, P. and Zdancewic, S.: *Advanced control flow in Java card programming*, LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, New York, NY, USA, ACM Press, pp. 165–174 (2004).
- [9] Microsoft: *XMLHttpRequest Object* (2007). <http://msdn2.microsoft.com/en-us/library/ms535874.aspx>.
- [10] mozilla.org: *Rhino: JavaScript for Java* (2006). <http://www.mozilla.org/rhino/>.
- [11] Shivers, O.: *Continuations and threads: Expressing machine concurrency directly in advanced languages*, Proceedings of the Second ACM SIGPLAN Workshop on Continuations, New York, NY, USA, ACM Press, pp. 2–1—2–15 (1997).
- [12] Stephenson, S.: *Prototype Javascript Library easing the development of dynamic web applications* (2006). <http://www.prototypejs.org/>.
- [13] Tarditi, D., Lee, P. and Acharya, A.: *No Assembly Required: Compiling Standard ML to C*, ACM Letters on Programming Languages and Systems, Vol. 1, No. 2, pp. 161–177 (1992).
- [14] Wand, M.: *Continuation-based multiprocessing*, LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming, New York, NY, USA, ACM Press, pp. 19–28 (1980).
- [15] 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一: オブジェクト指向並列言語 OPA のためのコード生成手法, 情報処理学会論文誌: プログラミング, Vol. 42, No. SIG 11 (PRO 12), pp. 1–13 (2001).