Transforming Javascript event-loop into a scalable pipeline

Etienne Brodu
etienne.brodu@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621,
Villeurbanne, France

Stéphane Frénot
stephane.frenot@insa-lyon.fr
Université de Lyon, INRIA,
INSA-Lyon, CITI-INRIA, F-69621,
Villeurbanne, France

Frédéric Oblé
frederic.oble@worldline.com
Worldline
Bât. Le Mirage, 53 avenue Paul
Krüger
69624 Villeurbanne Cedex

ABSTRACT

The development of a web application often starts with a feature-oriented approach allowing to quickly react to users feedbacks. However, this approach poorly scales in performance. Yet, the audience of a web application can increase by an order of magnitude in a matter of hours. This first approach is unable to deal with the higher connections spikes. It leads the development team to adopt a scalable approach often linked to new development paradigm such as dataflow programming. This represents a disruptive and continuity-threatening shift of technology. To avoid this shift, we propose to abstract the feature-oriented development into a more scalable high-level language. Indeed, reasoning on this high-level language allows to dynamically cope with audience growth and decrease.

We propose a compilation approach that transforms a Javascript, single-threaded web application into a network of small independent parts communicating by message streams. We named these parts *fluxions*, by contraction between a flow [1] and a function. The dynamic reorganization of these parts in a cluster of machine can help an application to deal with its load in a similar way network routers do with IP traffic. We evaluate this approach by applying the compiler to real web applications. We successfully transform a web application to parallelize the execution of an independent part and present the results.

Categories and Subject Descriptors

Software and its engineering [**Software notations and tools**]: Compilers—*Runtime environments*

General Terms

Compilation, dataflow, code transformation

Keywords

Flow programming, Web, Javascript

1. flux in french

1. INTRODUCTION

The growth of web platforms is partially due to Internet's capacity to allow very quick releases of a minimal viable product (MVP). In a matter of hours, it is possible to release a prototype and start gathering a user community. *"Release early, release often"*, and *"Fail fast"* are the punchlines of the web entrepreneurial community. It is crucial for the prosperity of such project to quickly validate that the proposed solution meets the needs of its users. Indeed, the lack of market need is the number one reason for startup failure [2]. That is why the development team quickly concretises an MVP using a feature-driven approach and iterates on it.

If the service successfully complies with users requirements, its community might grow with its popularity. The service needs to be scalable to be able to respond to this growth. However, feature-based development best practices are hardly compatible with the requirement of scalability. The features are bundled in modules which spread through the code base. This organization in modules overlaps and disturbs the organization of a scalable execution. Eventually this growth requires to discard the initial approach to adopt a more efficient processing model. Many of the most efficient models decompose applications into execution units, disregarding the initial feature oriented approach. This decomposition may be spread over a cluster of commodity machines [8]. MapReduce [6] and the Staged Event-driven Architecture (SEDA) [18] are famous examples of that trend. Once split, the service parts are connected by an asynchronous messaging system. Many tools have been developed to express and manage these service parts and their communications. We can cite Spark [20], MillWheel [2], Timestream [15], Naiad [12] and Storm [16], and many others. However, these tools are in disruption from the initial approach. It requires the development team either to be trained or to hire experts, and more importantly, to start over the initial code base. This shift causes the development team to spend development resources in background without adding visible value for the users. It is a risk for the evolution of the project as the number two and three reasons for startup failures are running out of cash, and missing the right competences[2].

The risks described above come from a disruption between the two levels of application expression, the feature level and the execution level. To lift these risks, we propose a tool to identify the alignment between the two levels, so as to allow a continuous transition from one to the other and back. We focus on web applications driven by users requests, develo-

2. `https://www.cbinsights.com/blog/startup-failure-post-mortem/`

ped in Javascript using the *Node.js* execution environment.

Javascript is increasingly used to develop web applications. It is the most used language on Github [3] and StackOverflow [4]. We think that it is possible to analyze this type of application as a stream of requests, passing through a pipeline of stages. Indeed, the event-loop used in *Node.js* is very similar to a pipeline architecture. We propose a compiler to transform a Javascript application into a network of autonomous parts communicating by message streams. We named these parts *fluxions*, by contraction between a flux and a function. We are interested in the problems arising from the isolation of the global memory into these fluxions. We present an early version of this tool as a proof of concept for this compilation approach. We start by describing in section 2 the execution environment targeted by this compiler. Then, we present the compiler in section 3, and its evaluation in section 4. We compare our work with related works in section 5. And finally, we conclude this paper.

2. FLUXIONAL EXECUTION MODEL

In this section, we present an execution model to provide scalability to web applications. To achieve this, the execution model provides a granularity of parallelism at the function level. Functions are encapsulated in autonomous execution containers with their state, so as to be reallocated and executed in parallel. This execution model is close to the actors model, as the execution containers are independent and communicate by messages. The communications are assimilated to stream of messages, similarly to the dataflow programming model. It allows to reason on the throughput of these streams, and to react to load increases.

In the following paragraphs, we present the autonomous containers : fluxions. Then we present the messaging system to carry the communication between these execution containers. Finally, we present an example application using this execution model.

2.1 Fluxions

In our fluxional execution model, an application is partitioned into parts encapsulated in autonomous execution containers named fluxions. A fluxion is composed of a unique name to receive messages, a processing function, and a local memory called a *context*. Figure 1 presents the fluxionnal language we designed to express fluxions. At a message reception, the fluxion modifies its *context*, and sends messages to downstream fluxions. The *context* handles the state on which a fluxion relies between two message receptions.

Message passing is unable [5] to replace the synchronization allowed by shared state. To allow this synchronization when required, the execution model allows fluxions to share state between their contexts. The fluxions that needs to synchronize together are grouped with the same tag. To assure the consistency of the shared state, all the fluxions of a group are executed sequentially. Though, the different groups of fluxions are executed in parallel.

2.2 Messaging system

The messaging system assures the stream communications between fluxions. It carries messages based on the names of

3. `http://githut.info/`
4. `http://stackoverflow.com/tags`
5. at reasonable costs

⟨program⟩ ⊨ ⟨flx⟩ | ⟨flx⟩ *eol* ⟨program⟩
⟨flx⟩ ⊨ `flx` ⟨id⟩ ⟨tags⟩ ⟨ctx⟩ *eol* ⟨streams⟩ *eol* ⟨fn⟩
⟨tags⟩ ⊨ `&` ⟨list⟩ | *empty string*
⟨streams⟩ ⊨ `null` | ⟨stream⟩ | ⟨stream⟩ *eol* ⟨streams⟩
⟨stream⟩ ⊨ ⟨op⟩ ⟨dest⟩ [⟨msg⟩]
⟨dest⟩ ⊨ ⟨list⟩
⟨ctx⟩ ⊨ `{`⟨list⟩`}`
⟨msg⟩ ⊨ `[`⟨list⟩`]`
⟨list⟩ ⊨ ⟨id⟩ | ⟨id⟩ , ⟨list⟩
⟨op⟩ ⊨ `>>` | `->`
⟨id⟩ ⊨ *Identifier*
⟨fn⟩ ⊨ *imperative language and stream syntax*

Figure 1: Syntax of a high-level language to represent a program in the fluxionnal form

the recipient fluxions. After the execution of a fluxion, it queues the resulting messages for the event loop to process.

The execution cycle of an example fluxional application is illustrated in figure 2. The source code for this application is in listing 1 and the fluxional code for this application is in listing 2. In the schema, circles represent registered fluxions. The fluxion *reply* has a context containing the variable `count`. The plain arrows represent the actual message paths in the messaging system, while the dashed arrows between fluxions represent the message streams as seen in the fluxionnal application.

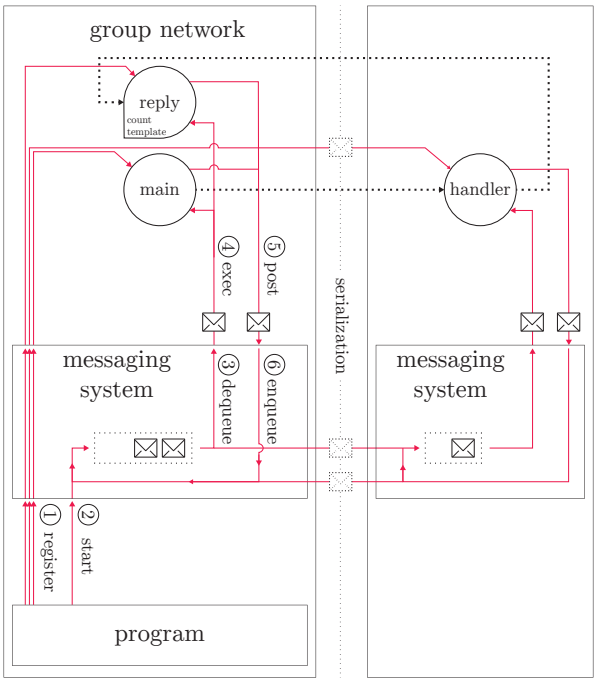

Figure 2: The fluxionnal execution model in details

The *main* fluxion is the first fluxion in the flow. It never receives any message, but when the application receives

a request, the *main* fluxion triggers the flow with a `start` message, ②. This first message is to be received by the next fluxion *handler*, ③ and ④. The fluxion *handler* sends back a message, ⑤, to be enqueued, ⑥. The system loops through steps ③ through ⑥ until the queue is empty. This cycle starts again for each new incoming request causing another `start` message.

2.3 Service example

To illustrate the fluxional execution model, and the compiler, we present an example of a simple web application. This application reads a file, and sends it back along with a request counter.

The original source code of this application is available in listing 1 [6]. In this source code, some points are worth noticing. The `handler` function, line 5 to 11, receives the input stream of request. The `count` variable at line 3 increments the request counter. This object needs to be persisted in the fluxion *context*. The `template` function simply formats the output stream to send back to the client. The `app.get` and `res.send` functions, respectively line 5 and 8, interface the application with the clients. And between these two interface functions is a chain of three functions to process the client requests : `app.get` ↠ `handler` → `reply`.

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      res.send(err || template(count, data));
9    });
10 });
11
12 app.listen(8080);
```

Listing 1: Example web application

This application is transformed into the high-level fluxional language in listing 2 which is illustred in Figure 2.

The application is organized as follow. The flow of requests is received from the clients by the fluxion `main`, it continues in the fluxion `handler`, and finally go through the fluxion `reply` to be send back to the clients. The fluxions `main` and `reply` have the tag `network`. This tag indicates their dependency over the network interface, because they received the response from and send it back to the clients. The fluxion `handler` has the tag `isolated`, it doesn't have any dependencies, hence it can be distributed on an isolated event-loop.

The last fluxion, `reply`, depends on its context to holds the variable `count` and the function `template`. It also depends on the variable `res` created by the first fluxion, `main`. This variable is carried by the stream through the chain of fluxion until the fluxion `reply` that depends on it. This variable holds the references to the network sockets. It is the variable the group `network` depends on.

Moreover, if the last fluxion, `reply`, did not had a context, the group `network` would be stateless. The whole group could be replicated as many time as needed.

This execution model allows to parallelize the execution of an application. Some parts are arranged in pipeline, like the fluxion `handler`, some other parts are replicated, like could be the group `network`. This parallelization improves the scalability of the application. Indeed, as a fluxion contains its state and expresses its dependencies, it can be migrated. It allows to adapt the number of fluxions per core to adjust the resource usage in function of the desired throughput.

Our goal, as described in the introduction, is not to propose a new programming paradigm with this high-level language but to automate the architecture shift. We present the compiler to automate this architecture shift in the next section.

```
1  flx main & network
2  >> handler [res]
3    var app = require('express')(),
4        fs = require('fs'),
5        count = 0;
6
7    app.get('/', >> handler);
8    app.listen(8080);
9
10 flx handler & isolated
11 -> reply [res]
12   function handler(req, res) {
13     fs.readFile(__filename, -> reply);
14   }
15
16 flx reply & network {count, template}
17 -> null
18   function reply(error, data) {
19     count += 1;
20     res.send(err || template(count, data));
21   }
```

Listing 2: Example application expressed in the high-level fluxional language

3. FLUXIONNAL COMPILER

The source languages we focus on should present higher-order functions and be implemented as an event-loop with a global memory. *Node.js* is an example of implementation for such a language. We developed a compiler that transforms a *Node.js* application into a fluxional system compliant with the architecture described in section 2. This compiler identifies application parts as fluxions to represent the application as a pipeline and resolves their memory dependencies to execute them in isolation of the global memory. As explained in the previous section, this isolation allows fluxions to be executed in parallel.

The compiler analyzes the Abstract Syntax Tree (AST) representing the source of an application. From the AST, it identifies asynchronous calls as rupture points. Section 3.1 defines rupture points, and explains how the compiler detects them. From the AST, it also builds a representation of the scopes of each variable used in the application to map the dependencies between the stages of the pipeline. Section 3.2 explains how the compiler distribute the central memory into isolated fluxions.

3.1 Analyzer

3.1.1 Rupture points

A rupture point is a call of a loosely coupled function. It is an asynchronous call without subsequent synchronization with the caller. In *Node.js*, I/O operations are asynchronous functions and indicates such rupture point between two application parts. The two application parts are the caller of the asynchronous function call on one hand, and the callback provided to the asynchronous function call on the other hand.

6. The listings are also available on github[5].

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with data not available in the caller context. We distinguish three kinds of callbacks.

Iterators are functions called for each item in a set, often synchronously.

Listeners are functions called asynchronously for each event in a stream.

Continuations are functions called asynchronously once a result is available.

There is two types of asynchronous callbacks : listeners and continuations. Similarly, there is two types of rupture point, respectively *start* and *post*.

Start rupture points are indicated by listeners. They are on the border between the application and the outside, continuously receiving incoming user requests. An example of a start rupture point is in listing 1, between the call to `app.get()`, and its listener `handler`. These rupture points indicate the input of a data stream in the program, and the beginning of a chain of fluxions to process this stream.

Post rupture points are indicated by continuations. They represent a continuity in the execution flow after an asynchronous operation yielding a unique result, such as reading a file, or querying a database. An example of a post rupture points is in listing 1, between the call to `fs.readFile()`, and its continuation `reply`.

The isolation of the execution between the asynchronous call and the callback is illustrated figure 3. The interface line represent the limit between two fluxions. It means that the upstream fluxion sends a message to the downstream fluxion to continue the execution with the callback.

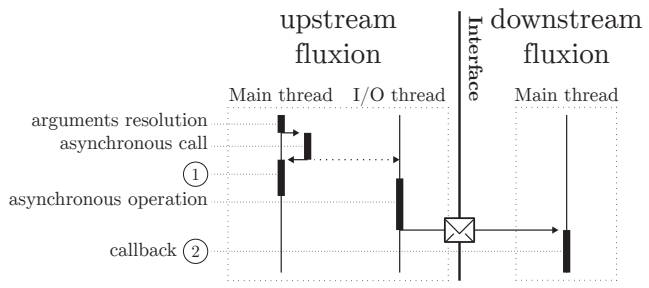

Figure 3: The rupture point interface is placed between the asynchronous operation and the callback in between the two call stacks.

3.1.2 Detection

Listeners and continuations are asynchronous because they are called asynchronously, and not because they are defined differently than any other function. Therefore, the identification of a rupture point holds on the callee, not on the callback. In listing 1, the two rupture points are identified because of `app.get` and `fs.readFile`, not because of `handler` and `reply`. Moreover, the asynchronism is provided by the execution engine, not the language. It is impossible to identify an asynchronous function from a synchronous function based on their syntax. The compiler uses a list of common asynchronous callee, like the `express` and file system methods. This list can be augmented to match asynchronous callee individually for any application. To identify the callee, the analyzer walks the AST to find a call expression matching this list.

After the identification of the callee, the callback needs to be identified as well to be encapsulated in the downstream fluxion. For each asynchronous call detected, the compiler test if one of the arguments is of type `function`. Some callback functions are declared *in situ*, and are trivially detected. For variable identifier, and other expressions, the analyzer tries to detect their type. To do so, the analyzer walks back the AST to track their assignations and modifications, and determine their last value.

Missing callbacks by false negatives in the detection is sub-optimal, but false positives are more critical, as they eventually introduce bugs. Therefore, the detection needs to be as accurate as possible to screen out false positives. The variable tracker is still in early development and is limited to only a few cases. In future works, our tracking method would be inspired from the points-to analysis [17].

3.2 Pipeliner

The rupture between two callbacks is not trivial. Languages providing higher-order functions often provide closures as the implementation of lexical scoping. A closure is a function that keeps the execution context of its initial definition. It means that the callback provided to an asynchronous function keeps access on the memory scope of the caller of this asynchronous function. To execute a callback in parallel, its memory needs to be independent from the global memory. The dependencies resulting from closures needs to be addressed, and resolved by the second part of our compiler, the pipeliner.

In a stream processing, there is roughly two kinds of usage of the global memory : data and state [7]. Naively, the data represent a communication channel between different application parts, and the state represents a communication channel between different instant in time. The data flow from stage to stage through the pipeline, and are never stored on any fluxion. In the source application, it is stored in the heap, only as a buffer between the different callbacks. The state, on the other hand, remains in the memory to impact the future behaviors of the application. State might be shared by several parts of the application. So, the identification of rupture points is not enough for a fluxion to be isolated, and its execution parallelized. The compiler also needs to analyze the memory accesses to identify which part of the state is needed by each fluxion, and allow their coordination.

3.2.1 Scope isolation

In Javascript, the memory is organized in scopes. They are nested one in the other up to the all-enclosing global scope. Each function creates a new scope containing variables local to itself. It is chained to the scope of the parent function, so that the child function can access variables in the scope of the parent functions, up to the global scope. However, the scope of the function inside a fluxion is isolated from its ancestors.

A rupture point eventually breaks a chain of scopes. When it is between a child scope and its parent, it makes the child unable to access its parent as expected. The parent is in the upstream fluxion, and the child in the downstream fluxion. Or when it is between a closure, and its definition context. The definition context is in the upstream fluxion and the closure in the downstream fluxion. If this situations

aren't resolved, they introduce errors in the compilation result. The linker analyzes how scopes are distributed among the fluxions to identify how the variables broken onto several fluxions are used in the upstreams and downstreams fluxions. At the end of this analysis, the compiler knows for every variable, if it is read or modified inside each fluxion.

However, scopes are an abstract representation of the memory, it is only the surface. Internally, the heap is a global memory without any fencing. A variable in one scope can point to the same object as another variable in another scope. If the first variable is modified, the modification propagates to the second variable, without this second variable being visibly modified. This situation produces side-effects between the two scopes. We call these side-effects scope leaking. If these two scopes are isolated on two different workers, the side-effects are unable to propagate as expected. We identified three basic situations leading to scope leaks.

Assignment.

If the object of a variable is assigned to another variable, there is possibly side effects between these two variables. It is illustrated in listing 3. The variable `a` is never modified visibly, yet, it is modified through the variable `b`.

```
var a = {item: 'unchanged'};
var b = a;

async_fn(function callback() {
  b.item = 'changed';
  console.log(a.item); // 'changed';
})
```

Listing 3: Example of a scope leak due to assignment

Function call.

When a variable containing an object is passed as an argument to a function call, it is assigned to a different variable as a parameter inside the function scope. There is possibly side effects between these two variables. It is illustrated in listing 4. The variable `a` is passed as an argument to the function `async_fn`. The function `callback` is then called by `async_fn` with the object from `a` as an argument. The variable `a` is never modified visibly, yet it is modified through the variable `b`.

```
var a = {item: 'unchanged'};

async_fn(function callback(b) {
  b.item = 'changed';
  console.log(a.item); // 'changed';
}, a);
```

Listing 4: Example of a scope leak due to a function call

Closure.

When a closure is called, it can modify its creation context from outside the scope of this creation context. It leads to side effects from outside this scope. It is illustrated in listing 5. The variable `a` is only modified visibly inside an anonymous function. This anonymous function is returned to be assigned in the variable `closure`. The variable `a` is modified when `closure` is called.

```
function closureFactory() {
  var a = {item: 'unchanged'};
  return function() {
    a.item = 'changed';
  }
}

var closure = closureFactory();

async_fn(function() {
  closure();
  // inside closureFactory : a.item === 'changed';
});
```

Listing 5: Example of a scope leak due to a closure

Our compiler is currently in early stage of development. It is unable to analyze the memory deeply enough to provide a sound and complete analysis. Hence, the static scope analysis previously presented is unable to take these scope leaking into account. We believe that with the right combination of static and dynamic analysis it is possible to produce a sound and complete representation of the memory. We leave the improvement of this analysis for future work.

3.2.2 State sharing

The previous analysis leads to a representation of the memory usage through the network of fluxions, and the dependencies between them. Depending on this representation, there is three different ways the compiler can resolve the dependencies broken by scopes isolation.

Scope.

If a variable is modified inside only one fluxion, then it can be part of the context of this fluxion. The fluxion has an exclusive access to it. If the context of a fluxion doesn't contains references shared with other fluxions, then it can be isolated on its own worker for its execution to be parallelized.

Stream.

If a variable is modified inside one fluxion, and read inside at least one downstream fluxion, then it can be part of the message to be sent to the downstream fluxions.

It is impossible to send variables to upstream fluxions, without race conditions. Indeed, if the fluxion retro propagates the variable for an upstream fluxion to read, the upstream fluxion might use the old version while the new version is on its way.

Additionally, it is currently impossible to stream variables containing closures. Indeed, it is impossible to serialize closure from within Javascript. As the fluxionnal execution model is currently confined inside the Javascript execution environment, it is unable to send closures from one worker to the other.

Share.

If a variable is needed for modification by more than one fluxion, or is read by an upstream fluxion, then it needs to be synchronized between the fluxions. The synchronization of a distributed memory is a well-known subject starting with the BASE semantics[8]. However, in this work, we choose to not allow such synchronization between workers. All the fluxions sharing a variable are gathered on the same worker to disallow parallel access on their shared memory. Similarly, if a fluxion shares references or closures with other fluxions, either in its context, or streams, they need to be hosted on the same worker.

4. EVALUATION

The goal of this evaluation is to prove the possibility for an application to be compiled in order to defer parts of its execution on a remote worker. We want to show the limitations of this isolation for future works, and the modifications needed to circumvent these limitations.

For brevity, we present in this paper only one test on a real application, gifsockets-server [7]. This application is part of the selection from our previous paper [4]. We chosed it because it is a complete, working, application, not a library, and it is simple enough to illustrate this evaluation.

This application is an example of a chat using gif-based communication channels. The client, a page containing a never-ending gif, sends a request containing a text typed by the user. The server transforms this text into a gif frame, and pushes this frame back to the never-ending gif to be displayed. Listing 6 is a simplified version of this application, containing only critical sections.

The web application framework used in this application, *express*, allows to register chains of functions to process user requests. On line 25, the call to `app.post` register two functions to process the requests on the url `/image/text`. The closure `saveBody`, line 7, returned by `bodyParser`, line 6, and the method `routes.writeTextToImages` from the external module `gifsockets-middleware`, line 3. The closure `saveBody` gather the whole request, and let *express* call the next function in the chain, `routes.writeTextToImages`, by calling `next`, line 20.

```
1  var express = require('express'),
2      app = express(),
3      routes = require('gifsockets-middleware');
4      getRawBody = require('raw-body');
5
6  function bodyParser(limit) {
7    return function saveBody(req, res, next) {
8      getRawBody(req, {
9        expected: req.headers['content-length'],
10       limit: limit
11     }, function (err, buffer) {
12       // If there was an error (e.g. bad length, over
             length), respond poorly
13       if (err) {
14         res.writeHead(500, {
15           'content-type': 'text/plain'
16         });
17         return res.end('Content was too long');
18       }
19       req.body = buffer;
20       next();
21     });
22   };
23 }
24
25 app.post('/image/text', bodyParser(1 * 1024 * 1024),
       routes.writeTextToImages);
26 app.listen(8000);
```

Listing 6: Simplified version of gifsockets-server

4.1 Compilation

We compile this application with the compiler detailed in section 3. The function call `app.post`, line 25, is asynchronous, but the compiler is currently unable to detect the function `saveBody` returned by `bodyParser` as a callback. The compiler detects only one rupture point, between `getRawBody` and its anonymous callback, line 11. It encapsulates this callback in a fluxion named `anonymous_1000`. The original callback is replaced with a placeholder function to send a message to this fluxion now containing the callback.

The compiler identifies that this callback uses the variables `req`, `res`, and `next`. It puts these variables in the stream of the message to send to the downstream fluxion `anonymous_1000`.

The compilation doesn't seem to introduce bugs, as the result of compilation executes without errors, and works as expected. However, it is important to note that the fluxion `anonymous_1000` is not yet isolated on a remote worker. Therefore, the variables used in the fluxion, `req`, `res` and `next`, are still shared with the rest of the application. Our goal is to isolate this fluxion in a different memory heap, to be able to safely parallelize its execution.

```
1  flx app_js
2  >> anonymous_1000 [req, res, next]
3    var express = require('express'),
4        app = express(),
5        routes = require('gifsockets-middleware');
6        getRawBody = require('raw-body');
7
8    function bodyParser(limit) {
9      return function saveBody(req, res, next) {
10       getRawBody(req, {
11         expected: req.headers['content-length'],
12         limit: limit
13       }, >> anonymous_1000);
14     };
15   }
16
17   app.post('/image/text', bodyParser(1 * 1024 * 1024),
         routes.writeTextToImages);
18   app.listen(8000);
19
20 flx anonymous_1000
21 -> null
22   function (err, buffer) {
23     // If there was an error (e.g. bad length, over
           length), respond poorly
24     if (err) {
25       res.writeHead(500, {
26         'content-type': 'text/plain'
27       });
28       return res.end('Content was too long');
29     }
30     req.body = buffer;
31     next();
32   }
```

Listing 7: Compilation result of the simplified version of gifsockets-server

4.2 Isolation

The variables `req` and `res` points to objects containing closures, and the variable `next` points to a closure. The fluxion `anonymous_1000` require access over these closures. Indeed, in listing 7, it modifies the attribute `body` of the object `req`, line 30, to store the text of the received request. It calls `next` to continue the execution, line 31. And in case of error, it uses the function `res.writeHead`, line 25, and `res.end`, line 28. It is impossible to serialize closures from within Javascript. Isolating the fluxion `anonymous_1000` produces runtime exceptions because it lacks access to these closures. We detail in the next paragraph, how the compiler handles this situation. In this evaluation, we ignore the case of error, and focus on the closure `next`.

4.2.1 *Closure next*

The function `next` is a closure over the *express* `Router`. This function is provided by the `Router` itself to allow one

7. https://github.com/twolfson/gifsockets-server

function in the chain to call the next. It is impossible to send this closure to the isolated fluxion. Instead, we modify *express*, so as to be compatible with the fluxionnal execution model.

The `req`, `res` and `next` objects needs to stay on the master worker to preserve their closures. The *express* `Router` register a local fluxion named `express_dispatcher` to holds these objects on the master worker, and receives the result of the isolated fluxion `anonymous_1000`. The application sends the original object to the fluxion `express_dispatcher` and serialized copies to the isolated fluxion `anonymous_1000`. In this latter fluxion, the anonymous callback do its computation ; it assigns the received `body` as an attribute of `req`.

In the original application, the anonymous callback finishes by calling the function `next` to let the `Router` call the next function to process the request. In the compiled application, this function `next` is not available on the isolated worker. Instead, the anonymous callback inside `anonymous_1000` calls a function `next` specially provided by the fluxionnal execution model to send a message to the fluxion `express_dispatcher` with the modified copies of `req` and `res`.

In the original application, *express* relies on side-effects on the objects `req` and `res` to get their modifications. The call to `next` doesn't need them as argument. In the isolated fluxion, as the serialized object and their originals are isolated from each other, side-effects don't propagate. The special `next` function needs explicit references to the modified objects to send them back to `express_dispatcher`. The fluxion `express_dispatcher` then merges back the modified copies and their originals, before calling the original function `next`.

After the modifications detailed above, the server works as expected for the subset of functionalities we modified. The isolated fluxion correctly receives, and returns its serialized messages. The client successfully receives a gif frame containing the text. However, in this evaluation, we ignored the case of error.

4.2.2 Fluxionnal web framework

In case of error, the anonymous callback calls `res.writeHead` and `res.end`. These two closures are similar to the closure `next`. It is possible to extend the modifications presented above to build a complete web application framework, with some limitations detailed below. Indeed, the evaluation proves that it is possible to modify the *express* framework to be compatible with the fluxionnal execution model.

The closure `next` is assured to be called only once at the end of the callback. It can be called asynchronously, and can be assimilated to a rupture point. Therefore, it is safe to replace it by a communication between the two workers. On the other hand, the functions `res.writeHead` and `res.end` are synchronous. It is unsafe to replace every call by a communication between the two workers. It would lead to race conditions. These calls needs to modify the serialized, local copies of `req` and `res`, and sends the result to the master only once.

5. RELATED WORKS

Most of the code transformation intended for parallelization require annotations, or transform loops inside of a sequential program. Our approach is different in that we aim at avoiding completely annotations. Indeed, the knowledge of asynchronism from the developer is enough so that we don't need more annotations. Furthermore, the transformation of the loops in sequential program is inherently limited by the sequential surrounding. Amdahl's law state that such program cannot expect high speedup due to parallelization. On the otherhand, the event-loop our approach is based on doesn't have any sequential surrounding. It is the most outer loops, because it is not part of the language, but of the runtime. Therefore, our approach is not limited by this sequential surrounding.

The idea to split a task into independent parts goes back to the Actor's model [9] in 1973, and to Functional programming, like Lucid [3] in 1977 and all the following works on DataFlow leading up to Flow-Based programming (FBP) and Functional Reactive Programming (FRP). Both FBP and FRP, recently got some attention in the Javascript community with the projects *NoFlo*[8], *Bacon.js*[9] and *react*[10].

The execution model we presented in section 2, is inspired by some works on scalability for very large system, like the Staged Event-Driven Architecture (SEDA) by Matt Welsh [18], System S developped in the IBM T. J. Watson research center [10, 19], and later the MapReduce architecture [6]. It also drew its inspiration from more recent work following SEDA. Among the best-known following works, we cited in the introduction Spark [20, 21], MillWheel [2], Timestream [15] and Storm [16]. The first part of our work stands upon these thorough studies. However, we believe that it is too difficult for common developers to express their algorithm into a network of independent parts communicating through messages. This belief motivated us to propose a compiler from an imparative programming model to these more scalable, distributed execution engines.

The transformation of an imperative programming model to be executed onto a parallel execution engine was recently addressed by Fernandez *et. al.* [7]. However, like in similar works [13, 14], the developer needs to manually specify the distribution of state. We believe the difficulties encountered by developers in concurrent programming models lies more in the distribution of states, than on the structuration of the algorithm. Indeed, developers seems to have little difficulties programming in an asynchronous concurrent programming model, like the Javascript event-loop. In such programming model, the memory is global but the algorithm is ripped into multiple, asynchronous steps. While the synchronous concurrent programming model, based on multi-threading and locks to assure the consistency of shared states, is known to be more difficult to apprehend by novice developers [1].

Our compiler uses the *estools* suite to parse, manipulate and generate source code from Abstract Syntax Tree (AST)[11]. It modifies AST, as described in [11]. The implementation of the analyzer might be inspired from the points-to analysis in future works [17]. Our implementation is based on the work by Ryan Dahl : *Node.js*[12], as well as on one of the best-known *Node.js* web framework : *Express*[13].

6. CONCLUSION

In this paper, we presented our work on a high-level language allowing to represent a web application as a network

8. `http://noflojs.org/`
9. `https://baconjs.github.io/`
10. `https://facebook.github.io/react/`
11. `https://github.com/estools`
12. `https://nodejs.org/`
13. `http://expressjs.com/`

of independent parts communicating by message streams. We presented a compiler to transform a *Node.js* web application into this high-level representation. To identify two independent parts, the compiler spots rupture points in the application, possibly leading to memory isolation and thus, parallelism. The compiler is still in early development, and is unable to soundly distribute memory. However, we proved it is possible to compile an application so that parts of its execution are parallelized, with minimum helps from the developer - only to identify the asynchronous calls. We also presented the execution model to operate an application expressed in our high-level language. This distributed approach allows code-mobility which may lead to a better scalability. We believe this high-level approach can enable the scalability required by highly concurrent web applications without discarding the familiar monolithic and asynchronous programming model used in *Node.js*.

Références

[1] A Adya, J Howell et M Theimer. "Cooperative Task Management Without Manual Stack Management." In : *USENIX Annual Technical Conference* (2002).

[2] T Akidau et A Balikov. "MillWheel : Fault-Tolerant Stream Processing at Internet Scale". In : *Proceedings of the VLDB Endowment 6.11* (2013).

[3] Edward A Ashcroft et William W Wadge. "Lucid, a nonprocedural language with iteration". In : *Communications of the ACM* 20.7 (1977), p. 519–526.

[4] E Brodu, S Frénot et F Oblé. "Toward automatic update from callbacks to Promises". In : *AWeS* (2015).

[5] Etienne Brodu. *flx-example*. doi : 10.5281/zenodo.11945.

[6] J Dean et S Ghemawat. "MapReduce : simplified data processing on large clusters". In : *Communications of the ACM* (2008).

[7] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki et Peter Pietzuch. "Making state explicit for imperative big data processing". In : *USENIX ATC* (2014).

[8] A Fox, SD Gribble, Y Chawathe, EA Brewer et P Gauthier. *Cluster-based scalable network services*. 1997.

[9] C Hewitt, P Bishop, I Greif et B Smith. "Actor induction and meta-evaluation". In : *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1973).

[10] N Jain, L Amini, H Andrade et R King. "Design, implementation, and evaluation of the linear road benchmark on the stream processing core". In : *Proceedings of the ...* (2006).

[11] J Jones. "Abstract syntax tree implementation idioms". In : *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)* (2003).

[12] F McSherry, R Isaacs, M Isard et DG Murray. "Composable Incremental and Iterative Data-Parallel Computation with Naiad". In : *Microsoft Research* (2012).

[13] C Mitchell, R Power et J Li. "Oolong : asynchronous distributed applications made easy". In : *Proceedings of the Asia-Pacific Workshop on ...* (2012).

[14] R Power et J Li. "Piccolo : Building Fast, Distributed Programs with Partitioned Tables." In : *OSDI* (2010).

[15] Z Qian, Y He, C Su, Z Wu et H Zhu. "Timestream : Reliable stream computation in the cloud". In : *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (2013).

[16] A Toshniwal et S Taneja. "Storm@ twitter". In : *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14* (2014).

[17] S Wei et BG Ryder. "State-sensitive points-to analysis for the dynamic behavior of JavaScript objects". In : *ECOOP 2014–Object-Oriented Programming* (2014).

[18] M Welsh, SD Gribble, EA Brewer et D Culler. *A design framework for highly concurrent systems*. 2000.

[19] KL Wu, KW Hildrum et W Fan. "Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S". In : *Proceedings of the 33rd ...* (2007).

[20] M Zaharia et M Chowdhury. "Spark : cluster computing with working sets". In : *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010).

[21] M Zaharia, T Das, H Li, S Shenker et I Stoica. "Discretized streams : an efficient and fault-tolerant model for stream processing on large clusters". In : *Proceedings of the 4th ...* (2012).