Transforming Javascript event-loop into a scalable pipeline

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 [6]. MapReduce [4] and the Staged Event-driven Architecture (SEDA) [16] 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 [18], MillWheel [1], Timestream [13], Naiad [10] and Storm [14], 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. The contribution of this paper is the resolution of these problems, allowing the compilation. 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.

The fluxional execution model executes programs written in our high-level fluxionnal language, whose grammar is presented in figure 1. An application ⟨program⟩ is partitioned into parts encapsulated in autonomous execution containers named *fluxions* ⟨flx⟩. In the following paragraphs, we present the *fluxions*. Then we present the messaging system to carry the communications between *fluxions*. Finally, we present an example application using this execution model.

2.1 Fluxions

A *fluxion* ⟨flx⟩ is named by a unique identifier ⟨id⟩ to receive messages, and might be part of one or more groups indicated by tags ⟨tags⟩. A *fluxion* is composed of a processing function ⟨fn⟩, and a local memory called a *context* ⟨ctx⟩. At a message reception, the *fluxion* modifies its *context*, and sends messages on its output streams ⟨streams⟩ to downstream *fluxions*. The *context* handles the state on which a *fluxion* relies between two message receptions. In addition to message passing, the execution model allows *fluxions* to communicate by sharing state between their *contexts*. The fluxions that needs to synchronize together are grouped with the same tag, and loose their independence.

There is two types of streams, *start* and *post*, which corresponds to the nature of the rupture point yielding the stream. We differentiates the two types with two different arrows, double arrow ($\twoheadrightarrow$ or `>>`) for *start* rupture points and simple arrow ($\rightarrow$ or `->`) for *post* rupture points. The two different type of rupture points are further detailed in section 3.1.1.

3. `http://githut.info/`
4. `http://stackoverflow.com/tags`

⟨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⟩	⊨	⟨type⟩ ⟨dest⟩ [⟨msg⟩]
⟨dest⟩	⊨	⟨list⟩
⟨ctx⟩	⊨	`{`⟨list⟩`}`
⟨msg⟩	⊨	`[`⟨list⟩`]`
⟨list⟩	⊨	⟨id⟩ | ⟨id⟩ , ⟨list⟩
⟨type⟩	⊨	`>>` | `->`
⟨id⟩	⊨	*Identifier*
⟨fn⟩	⊨	*imperative language and stream syntax*

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

2.2 Messaging system

The messaging system assures the stream communications between fluxions. It carries messages based on the names of 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 figure 2, 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.

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[5]. 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 inter-

5. The listings are also available on github[3].

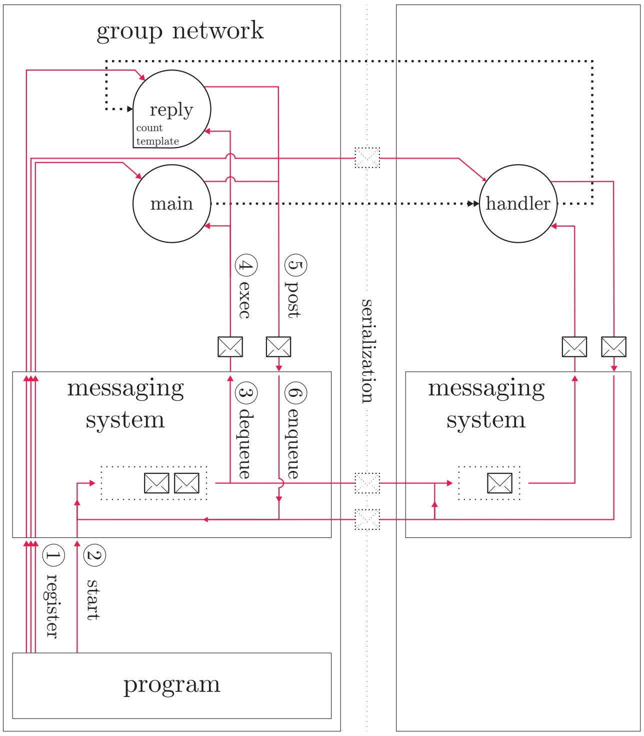


Figure 2: The fluxionnal execution model in details

face functions is a chain of three functions to process the client requests : `app.get` ↠ `handler` → `reply`.

```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename, function reply(err, data) {
    count += 1;
    res.send(err || template(count, data));
  });
});

app.listen(8080);
```

Listing 1: Example web application

This application is transformed into the high-level fluxionnal language in listing 2 which is illustrated 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.

```
flx main & network
>> handler [res]
  var app = require('express')(),
      fs = require('fs'),
      count = 0;

  app.get('/', >> handler);
  app.listen(8080);

flx handler & isolated
-> reply [res]
  function handler(req, res) {
    fs.readFile(__filename, -> reply);
  }

flx reply & network {count, template}
-> null
  function reply(error, data) {
    count += 1;
    res.send(err || template(count, data));
  }
```

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. Javascript is such a language : it doesn't require an event-loop implementation, but it is often implemented that way. *Node.js* is an example of such an implementation. We developed a compiler that transforms a *Node.js* application into a fluxional application compliant with the execution model described in section 2.

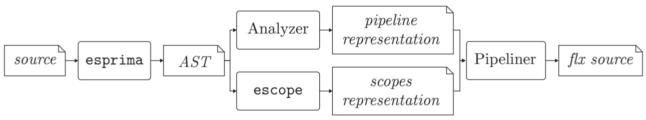


Figure 3: Compilation chain

The chain of compilation is described in figure 3. From the source of a *Node.js* application, the compiler extracts an Abstract Syntax Tree (AST). From this AST, the analyzer step identifies the limits of the different application parts and how they relate to form a pipeline. This first step outputs a pipeline representation of the application. Section 3.1 explains this first compilation step. In the pipeline representation, the stages are not yet independent and encapsulated into fluxions. From the AST, `escope` produces a representation of the memory scopes. The pipeliner step analyzes the pipeline representation and the scopes representation to distribute the shared memory into independent groups of fluxions. Section 3.2 explains this second compilation step.

3.1 Analyzer step

The limit between two application parts is defined by a rupture point. The analyzer identifies these rupture points, and outputs a representation of the application in a pipeline form, with application parts as the stages, and rupture points as the message streams of this pipeline.

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. Figure 4 shows an example of a rupture point with the execution of the two application parts isolated into fluxions. 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.

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, but only two are asynchronous : listeners and continuations. Similarly, there is two types of rupture points, 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`.

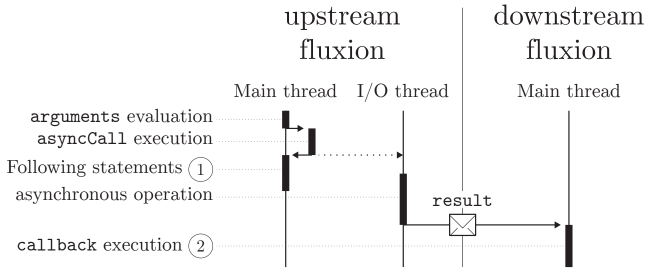
3.1.2 *Detection*

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.

3.2 Pipeliner step

A rupture point eventually breaks the chain of scopes between the upstream and downstream fluxion. The closure




```
asyncCall(arguments, function callback(result){ 2 });
// Following statements 1
```

Figure 4: Rupture point interface

in the downstream fluxion cannot access the scope in the upstream fluxion as expected. The pipeliner step replaces the need for this closure, allowing application parts to rely only on independent memory stores and message passing. It determines the distribution using the scope representation, which represents the variables dependencies between application parts. Depending on this representation, there is three different ways the compiler can replace the broken closure. We present these three alternative with the example figure 5.

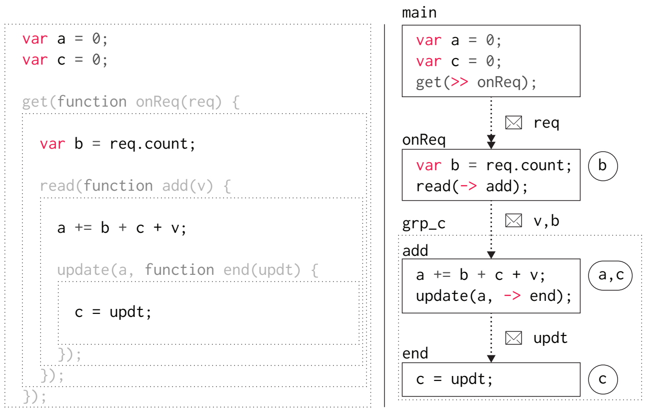


Figure 5: Variable management from Javascript to the high-level fluxionnal language

Scope.

If a variable is modified inside only one application part in the current *post* chain, then the pipeliner adds it the context of its fluxion.

In figure 5, the variable `a` is updated in the function `add`. The pipeliner steps store this variable in the context of the fluxion `add`.

Stream.

If a variable is modified inside an application part, and read inside downstream application parts, then the pipeliner makes the upstreap fluxion add this variable to the message stream to be sent to the downstream fluxions. It is impossible to send variables to upstream fluxions, without race conditions. 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.

In figure 5, the variable `b` is set in the function `onReq`, and read in the function `add`. The pipeliner steps makes the fluxion `onReq` send the updated variable `b`, in addition to the variable `v`, in the message sent to the fluxion `add`.

Exceptionally, if a variable is defined inside a *post* chain, then this variable can be streamed inside this *post* chain without restriction on the order of modification and read. Indeed, the execution of the upstream fluxion for the current *post* chain is assured to end before the execution of the downstream fluxion. Therefore, no read of the variable by the upstream fluxion happen after the modification by the downstream fluxion. The variable `b` is such a variable.

Share.

If a variable is needed for modification by several application parts, or is read by an upstream application part, 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[6]. To respect the semantics of the source application, we cannot tolerate inconsistencies. Therefore, the pipeliner groups all the fluxions sharing this variable within a same tag. And it adds this variable to the contexts of each fluxions.

In figure 5, the variable `c` is set in the function `end`, and read in the function `add`. As the fluxion `add` is upstream of `end`, the pipeliner steps groups the fluxion `add` and `end` with the tag `grp_c` to allow the two fluxions to share this variable.

4. REAL CASE TEST

The goal of this evaluation is to prove the possibility for an application to be compiled into a network of independent parts. We want to show the current limitation of this isolation and the modification needed on the application to circumvent these limitations.

For brevity, we present in this paper only one test on a real application, gifsockets-server [6]. This application was selected from the `npm` registry because it depends on `express`, it is tested and working, and it simple enough to illustrate this evaluation. It is part of the selection from our previous paper [citation removed for double-blind review].

This application is an example of a real-time chat using gif-based communication channels. The client 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 a never-ending gif to be displayed on the client. Listing 3 is a simplified version of this application, containing only essential lines.

On line 25, the application registers two functions to process the requests received 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` calls the asynchronous function `getRawBody` to get the request body. Its callback handles the errors, and calls `next` to continue the execution with the next function, `routes.writeTextToImages`.

```
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 3: Simplified version of gifsockets-server

6. `https://github.com/twolfson/gifsockets-server`

4.1 Compilation

We compile this application with the compiler detailed in section 3. The function call `app.post`, line 25, is a rupture point. However, its callbacks, `bodyParser` and `routes.writeTextToImages` are evaluated as function only at runtime. For this reason, the compiler ignores this rupture point, to avoid interfering with the evaluation.

The compilation result is in listing 4. The compiler detects a rupture point : the function `getRawBody` and its anonymous callback, line 11. It encapsulates this callback in a fluxion named `anonymous_1000`. The callback is replaced with a stream placeholder to send the message stream to this downstream fluxion. The variables `req`, and `next` are append to this message stream, to propagate their value from the `main` fluxion to the `anonymous_1000` fluxion.

When `anonymous_1000` is not isolated from the `main` fluxion, the compilation result works as expected. The variables used in the fluxion, `req` and `next`, are still shared between the two fluxions. Our goal is to isolate the two fluxions, to be able to safely parallelize their executions.

```
1  flx main
2  >> anonymous_1000 [req, 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     req.body = buffer;
24     next();
```

```
25 }
```

Listing 4: Compilation result of gifsockets-server

4.2 Isolation

In listing 4, the fluxion `anonymous_1000` modifies the object `req`, line 23, to store the text of the received request,and it calls `next` to continue the execution, line 24. These operations produce side-effects that should propagate in the whole application, but the isolation prevent this propagation. Isolating the fluxion `anonymous_1000` produces runtime exceptions. We detail in the next paragraph, how we handle this situation to allow the application to be compiled. This evaluation highlights the current limitations of the compiler, and present future works around them.

4.2.1 Variable `req`

The variable `req` is read in fluxion `main`, lines 10 and 11. Then it is associated in fluxion `anonymous_1000` to `buffer`, line 23. The compiler is unable to identify further usages of this variable. However, the side effect resulting from this association impacts a variable in the scope of the next callback, `routes.writeTextToImages`. We modified the application to explicitly propagate this side-effect to continue the execution with the function `next`.

4.2.2 Closure `next`

The function `next` is a closure provided by the `Router` to continue the execution with the next function to handle the client request. Because it indirectly relies on network sockets, it is impossible to isolate its execution with the `anonymous_1000` fluxion. Instead, we modify *express*, so as to be compatible with the fluxionnal execution model. We explain the modification below, and illustrate them in listing 5.

Originally, the function `next` is the continuation to allow the anonymous callback, to continue the execution with the next function to handle the request. To isolate the anonymous callback, this function is replaced on both ends. The *express* `Router` register a fluxion named `express_dispatcher` to continue the execution after the fluxion `anonymous_1000`. This fluxion is in the same group as the `main` fluxion, hence it has access to network sockets and to the original variable `req`. The original `next` function in the anonymous callback is replaced by a placeholder to push the stream to the fluxion `express_dispatcher`. The fluxion `express_dispatcher` receives the stream from the upstream fluxion `anonymous_1000`, merges back the modification in the variable `req`, before calling the original function `next` to continue the execution.

```
1  flx main
2  >> anonymous_1000 [req, 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 -> express_dispatcher
22   function (err, buffer) {
23     req.body = buffer;
24     -> express_dispatcher;
25   }
26
27 flx express_dispatcher
28 -> null
29   merge(req, msg.req);
30   next();
```

Listing 5: Simplified modification on the compiled result

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.

4.3 Future works

We intend to improve the compiler to evaluate the idea on a broader set of applications, without any manipulations required from the developer. Furthermore, at the light of this limitations and to circumvent them, we intend to push the compilation process during the execution. This just-in-time compilation process allows the compiler to detect callbacks dynamically evaluated. It also allows to analyze the memory to identify side-effects propagations, instead of relying only on the source code. And finally, this memory analysis allows closure serialization required to compile application using higher-order functions.

5. RELATED WORKS

The idea to split a task into independent parts goes back to the Actor's model [7] in 1973, and to Functional programming, like Lucid [2] 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*[7], *Bacon.js*[8] and *react*[9].

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 [16], System S developped in the IBM T. J. Watson research center [8, 17], and later the MapReduce architecture [4]. It also drew its inspiration from more recent work following SEDA. Among the best-known following works, we cited in the introduction Spark [18, 19], MillWheel [1], Timestream [13] and Storm [14]. The first part of our work stands upon these thorough studies. However, we believe that it is too difficult for most developers to distribute the state of an application. 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.* [5]. However, like in similar works [11, 12], the developers need to annotate their code to specify the distribution of state. It doesn't improve the

7. `http://noflojs.org/`
8. `https://baconjs.github.io/`
9. `https://facebook.github.io/react/`

accessibility of these distributed execution engines. Our approach avoids the need for annotations from the developer, thus aims at improving furthermore the accessibility.

Most approaches of imperative code transformation for parallelization focus on parallelizing loops in a sequential program. Because of this sequential structure, the speedup of parallelization is inherently limited. On the other hand, our approach is based on an asynchronous programming model, discarding this sequential structure. Hence the attainable speedup is not limited by sequential structure.

Our compiler uses the *estools* suite to parse, manipulate and generate source code from Abstract Syntax Tree (AST) [10]. It modifies AST, as described in [9]. The implementation of the analyzer might be inspired from the points-to analysis in future works [15]. Our implementation is based on the work by Ryan Dahl : *Node.js* [11], as well as on one of the best-known *Node.js* web framework : *Express* [12].

6. CONCLUSION

In this paper, we presented our work on a high-level language allowing to represent a web application as a network 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] T Akidau et A Balikov. “MillWheel : Fault-Tolerant Stream Processing at Internet Scale”. In : *Proceedings of the VLDB Endowment 6.11* (2013).

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

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

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

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

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

[7] 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).

[8] 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).

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

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

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

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

[13] 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).

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

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

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

[17] 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).

[18] 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).

[19] 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).

10. https://github.com/estools
11. https://nodejs.org/
12. http://expressjs.com/