Toward a compiler providing pipeline parallelism for the Javascript event-loop.

Etienne Brodu
etienne.brodu@insa-lyon.fr
CITI - INSA-Lyon
6 Av. des Arts
69621 Villeurbanne Cedex

Stéphane Frénot
stephane.frenot@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

Frédéric Oblé
frederic.oble@worldline.com
Worldline
Bât. Le Mirage
53 avenue Paul Krüger
CS 60195
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 represent a disruptive and continuity-threatening shift of technology. To avoid this shift, we propose to abstract the feature-oriented development into a high-level language, allowing a high-level code reasoning. This reasoning allows code mobility so as to dynamically cope with audience growth and decrease.

We propose a compiler that transforms a Javascript, monolithic, web application into a network of small independent parts communicating by message streams. We evaluate the approach by applying this compiler to a real web application. We succssefully transform a web application to parallelize the execution of an independent part. We named these parts *fluxions*, by contraction between a flux 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.

Categories and Subject Descriptors

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

General Terms

Compilation

Keywords

Flow programming, Web, Javascript

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 around. *"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. [1] That is why the development team quickly concretises an MVP and iterates on it using a feature-driven, monolithic approach. Such as proposed by imperative languages like Java or Ruby.

If the service successfully complies with users requirements, its community might grow with its popularity. If the service can quickly respond to this growth, it is scalable. However, it is difficult to develop scalable applications with the feature-driven approach mentioned above. Eventually this growth requires to discard the initial monolithic approach to adopt a more efficient processing model instead. Many of the most efficient models distribute the system on a cluster of commodity machines [8]. MapReduce [6] and the Staged Event-driven Architecture (SEDA) [19] 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 [21], MillWheel [2], Timestream [16], Naiad [13] and Storm [17], and many others. However, these tools impose specific interfaces and languages, different from the initial monolithic approach. It requires the development team either to be trained or to hire experts, and 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.

To lift the risks described above, we propose a tool to compile the initial code base into a high-level language com-

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

patible with the more efficient processing model. We focus on web applications driven by users requests, developed in Javascript using the *Node.js* execution environment.

Javascript is increasingly used to develop web applications. It is the most used language on Github [2], and the second one on StackOverflow [3]. 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 monolithic 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

The compiler we present in section 3 focus on web applications that tend to follow the functional paradigm while keeping a global memory. Such applications are built using functions that are executed sequentially to assure the exclusivity of access on the global memory. This is a serious performance issue, as it avoids to leverage the parallelism of modern architectures.

We present in this section a different execution model that isolate the memory accessible to some functions. This approach allows to execute these functions in parallel, hence, to benefit of the performance improvements of this parallelism. This execution model is close to the actor model, as the function are executed on autonomous execution unit with their own isolated memory, communicating by messages. Because we focus on real-time web applications, we insist on the streaming nature of these communications. The execution units exchange streams of messages, that correspond with the input stream of requests of the web application.

2.1 Fluxions and workers

The fluxional execution model manages and invokes autonomous execution units named fluxion ⟨flx⟩. A fluxion is composed of a unique name ⟨id⟩, a processing function ⟨fn⟩, and a persisted memory called a *context* ⟨ctx⟩. Its function ⟨fn⟩ consumes an input stream ⟨stream⟩ and generates one or more outputs streams to other fluxions ⟨dest⟩. The *context* persists the state on which a fluxion relies between two message receptions. At a message reception, the fluxion modifies its *context*, and sends back messages to downstream fluxions. A message is composed of the recipient fluxions' names and a body.

Fluxions are executed on workers. A worker is an event-loop and an isolated heap ; it is a *Node.js* instance. The context of a fluxion is lexically isolated. It has a distinct lexical scope containing variables not shared with any other fluxion. However, fluxions on the same worker share the same event-loop, and the same heap ; they can send references to each over. Fluxions on different workers have different event-loops and heaps ; their communications are serialized, so it is useless to send heap references. Fluxions are the stages in a pipeline architecture. The streams of messages between fluxions are carried by the messaging system.

The event-loop assures the exclusivity of operations on the heap. Only one fluxion is executed at once on a worker. Consequently, the more fluxions share states, the less time fraction each fluxion has for its execution. If a fluxion has its own exclusive state, it can be parallelized.

We represent here the syntax of a high-level language to represent a program in the fluxionnal form. It is the target for our compiler.

⟨program⟩ ⊨ ⟨flx⟩ | ⟨flx⟩ *eol* ⟨program⟩
⟨flx⟩ ⊨ `flx` ⟨id⟩ ⟨ctx⟩ ⟨worker⟩ *eol* ⟨streams⟩ *eol* ⟨fn⟩
⟨worker⟩ ⊨ `on` ⟨id⟩ | *empty string*
⟨streams⟩ ⊨ `null` | ⟨stream⟩ | ⟨stream⟩ *eol* ⟨streams⟩
⟨stream⟩ ⊨ ⟨op⟩ ⟨dest⟩ [⟨msg⟩]
⟨dest⟩ ⊨ ⟨list⟩
⟨ctx⟩ ⊨ `{`⟨list⟩`}`
⟨msg⟩ ⊨ `[`⟨list⟩`]`
⟨list⟩ ⊨ ⟨id⟩ | ⟨id⟩ , ⟨list⟩
⟨op⟩ ⊨ `>>` | `->`
⟨id⟩ ⊨ *Javascript identifier*
⟨fn⟩ ⊨ *Javascript and stream syntax*

2.2 Messaging system

In a distributed approach, the messages between fluxions would be carried over a distributed message broker. However this execution model is only a simulation of a distributed execution environement. We simplify the distributed message broker with a master message queue to centralize communication between workers, though, each worker has its own local message queue. The messaging system sends messages to the worker hosting the destination fluxion. Locally, the master worker hosts fluxions that need access to the external network or the global memory.

The execution cycle of a fluxional application is illustrated in figure 1. 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 lines between fluxions represent the message streams as seen in the fluxionnal application. The streams between workers are serialized.

The messaging system carries messages based on the names of the recipient fluxions. If two fluxions share the same name, it would lead to a conflicting situation for the messaging system. Every fluxion needs to be registered with a unique name. This registration associates a processing function with a unique name and an initial *context*. The registration is done by calling `register(<name>, <fn>, <context>)`, ①.

When a new request is received, a `start` message triggers the flow using the function `start(<msg>)`, ②. This first message represent the incoming of a request from a user. The system dequeues this message and dispatch it to the destination fluxion, *handler*, ③ and ④. The fluxion *handler* sends back a message using the function `post(<msg>)`, ⑤, to be enqueued in the centralized message queue, ⑥. The system loops through steps ③ and ④ until the queue is

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

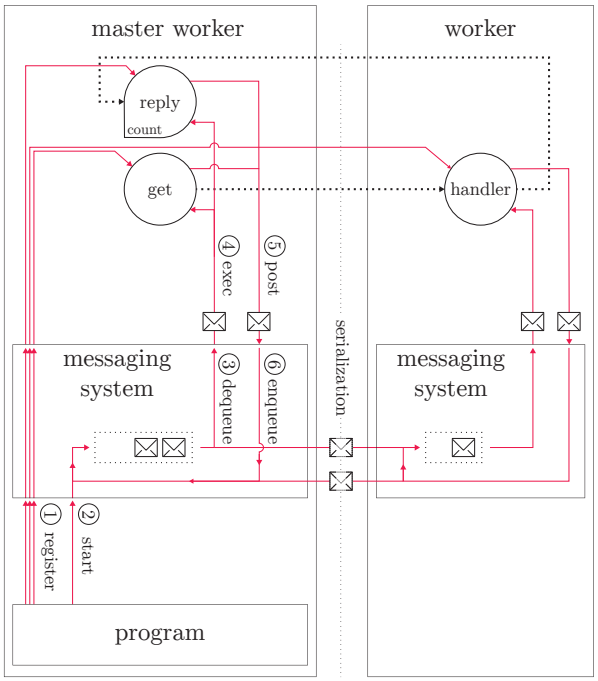

Figure 1: The fluxionnal execution model in details

empty. This cycle starts again for each new incoming request causing a `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 the file containing its own source code, and sends it back along with a request counter.

The original source code of this application is available on github[5], and in listing 1. In this source code, some points are worth noticing. The `handler` function, line 5 to 11, receives the input stream of request. It is the first function we execute in an isolated fluxion. The `count` variable at line 3 increments the request counter. This object needs to be persisted in the fluxion *execution context*. The `app.get` and `app.send` methods, respectively line 5 and 9, interface the application with the clients. The processing chain of functions occurs between these two functions : `get` ↠ `handler` → `readFile` → `reply` → `send`.

```
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: Simple web application that replies to every request with its own source code and a counter

This application is transformed manually into the high-level fluxionnal language in listing 2, and illustred in Figure 1. We expect a similar result with the compiler described in section 3.

```
1  flx get
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 on worker
11 -> reply [res]
12   function handler(req, res) {
13     fs.readFile(__filename, -> reply);
14   }
15
16 flx reply {count, template}
17 -> null
18   function reply(error, data) {
19     count += 1;
20     res.send(err || template(count, data));
21   }
```

Listing 2: Manual transformation of the example application in our high-level fluxional language

The application is organized as follow :

— The `get` fluxion is the *root* fluxion. It initializes the application to listen for user requests by calling `app.get`. Every request is forwarded on the stream to the `handler` fluxion, line 7.

— The `handler` fluxion reads the file containing the source code of the application, and forwards the result to the `reply` fluxion, line 13.

— The `reply` fluxion increments the counter, line 19, formats the reply, and sends it back to the user using the function `res.send`, line 20.

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 with a compiler. We present this compiler in the next section.

3. FLUXIONNAL COMPILER

Web applications are currently, mostly written in Java. The langage proposes both data encapsulation in objects and a threading model that allows the development of parallel applications. But, this approach is error-prone, and leads to deadlocks and other synchronization problems [1]. Since 2009, *Node.js* proposes an alternative to this model. It provides a Javascript execution environment for real-time web applications, with data encapsulation in modules, and a concurrency model based on an event-loop. We focus on this promising environment for its initial simplicity and efficiency.

An event-loop executes a program on a single thread of execution to avoid synchronization over the global memory, and the problems that come with it. It features asynchronous programming to avoid wasting execution time waiting for long operations to complete, like input/output operations. These operations are executed in parallel, as they don't require access to the global memory. To resume the execution the call to an asynchronous operation requires a callback. That is a function passed as a parameter of the callee to resume execution once the asynchronous operation is finished. Callbacks is a software construct present in any language with higher-order functions.

Callbacks are queued after the operation complete to be executed sequentially by the event-loop. A callback is loosely coupled with the caller of the asynchronous operation. They are not executed on the same call stack. This rupture marks out the separation between two independent application parts. The two parts are callbacks, as the caller of the asynchronous operation is itself the callback of another caller, up until the root of the program. The execution is organized as a tree of callbacks executed independently. It is a particularity of using the event-loop as a base for the concurrency model in the implementation of the execution engine.

However, 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 our compiler.

To summarize, the source languages we focus on should present higher-order functions and be implemented as an event-loop with a global memory. We develop a compiler that transforms a *Node.js* application into a fluxional system compliant with the architecture described in section 2. This compiler identifies rupture points between application parts and resolves their memory dependencies to execute them in isolation of the global memory, hence in parallel inside fluxions.

The compiler analyzes the Abstract Syntax Tree (AST) representing the source of an application. From the AST, it identifies rupture points between the callbacks to represent the application as a pipeline. Section 3.1 define 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.

We do not target all Javascript Web-based application as this work is only a proof of concept for the compilation. Our goal is to compile a few real applications without modifying their code, so as to validate this approach.

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.

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 2. 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.

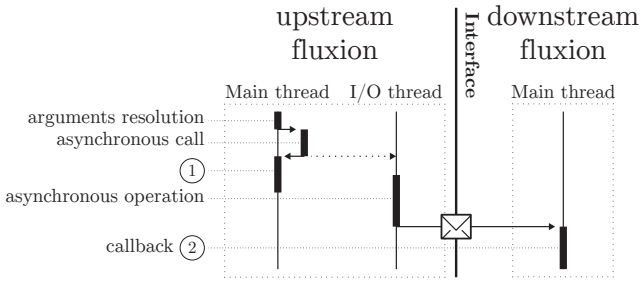

Figure 2: The rupture point interface is placed between the asynchronous operation and the callback in between the two call stacks.

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`. The asynchronism is provided by the execution engine, not the language. Therefore, 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.

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 compiler tries to detect their type by tracking their assignations and modification. 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. It walks an intermediate representation of the source code to spot the statements modifying a variable. From this intermediate representation, the variable tracker builds a dependency graph which helps the analyzer to detect the type of a variable at a certain point in the execution. 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 [18].

3.2 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 point in the application space, 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 a variable is assigned the object of 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.

A closure is a function which conserves its access over its creation context. When the closure is called, it can modify this creation context from outside the scope of this creation context. There is possibly 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. The scope analysis previously presented is unable to take these scope leaking into account. It is unable to analyze the memory deeply enough to provide a sound and complete analysis. Therefore it might lead to runtime errors. Moreover, we present only basic situations of scope leaking. Javascript exposes many other features leading to scope leaking, like prototype inheritance.

3.2.2 State sharing

Depending on the result of the previous analysis on the variable usage through scopes, there is three different ways the compiler can resolve the conflict.

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, but read inside at least one downstream fluxion, then it can be part of the message to be sent to the downstream fluxions.

It is possible to stream variables only to downstream fluxions. 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. To avoid such race conditions, we avoid retro propagation.

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, with Brewer's conjecture [9], and the BASE semantics[8]. However, we currently 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 [4]. 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.get` is asynchronous, but the compiler is 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, {
```

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

```
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 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

The idea to split a task into independent parts goes back to the Actor's model [10] 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*[5], *Bacon.js*[6] and *react*[7].

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 [19], System S developped in the IBM T. J. Watson research center [11, 20], 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 [21, 22], MillWheel [2], Timestream [16] and Storm [17]. 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

5. `http://noflojs.org/`
6. `https://baconjs.github.io/`
7. `https://facebook.github.io/react/`

addressed by Fernandez *et. al.* [7]. However, like in similar works [14, 15], 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) [8]. It modifies AST, as described in [12]. The implementation of the analyzer might be inspired from the points-to analysis in future works [18]. Our implementation is based on the work by Ryan Dahl : *Node.js* [9], as well as on one of the best-known *Node.js* web framework : *Express* [10].

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] 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] S Gilbert et N Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In : *ACM SIGACT News* (2002).

[10] 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).

[11] 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).

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

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

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

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

[16] 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).

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

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

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

[20] 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).

[21] 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).

[22] 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).

8. https://github.com/estools
9. https://nodejs.org/
10. http://expressjs.com/