Transforming Javascript Event-Loop Into a Pipeline

Etienne Brodu, Stéphane Frénot, Frédéric Oblé

Transforming Javascript Event-Loop Into a Pipeline

Etienne Brodu, Stéphane Frénot
{etienne.brodu, stephane.frenot}@insa-lyon.fr
Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, 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 real-time web application often starts with a feature-driven approach allowing to quickly react to users feedbacks. However, this approach poorly scales in performance. Yet, the user-base can increase by an order of magnitude in a matter of hours. This first approach is unable to deal with the highest connections spikes. It leads the development team to shift to a scalable approach often linked to new development paradigm such as dataflow programming. This shift of technology is disruptive and continuity-threatening. To avoid it, we propose to abstract the feature-driven development into a more scalable high-level language. Indeed, reasoning on this high-level language allows to dynamically cope with user-base size evolutions.

We propose a compilation approach that transforms a Javascript, single-threaded real-time 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 independence of these parts allows their execution to be parallel, and to organize an application on several processors to cope with its load, in a similar way network routers do with IP traffic. We test this approach by applying the compiler to a real web application. We transform this application to parallelize the execution of an independent part and present the result.

CCS Concepts

•Software and its engineering → Source code generation; Runtime environments;

Keywords

Flow programming; Web; Javascript

[1]flux in french

SAC 2016, April 04 - 08, 2016, Pisa, Italy

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-3739-7/16/04…$15.00

DOI: http://dx.doi.org/10.1145/2851613.2851745

1. INTRODUCTION

"Release early, release often", *"Fail fast"*. The growth of a real-time web service is partially due to Internet's capacity to allow very quick releases of a minimal viable product (MVP). It is crucial for the prosperity of such project to quickly validate that it meets the needs of its users. Indeed, misidentifying the market needs is the first reason for startup failure[2]. Hence the development team quickly concretizes an MVP using a feature-driven approach and iterates on it.

The service needs to be scalable to be able to respond to the growth of its user-base. However, feature-driven development best practices are hardly compatible with the required parallelism. The features are organized in modules which disturb the organization of a parallel execution [6, 12, 16]. Eventually the 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 [10, 20, 7]. However, these tools are in disruption from the initial approach. This shift causes the development team to spend development resources in background to start over the initial code base, without adding visible value for the users. It is a risk for the evolution of the project. Running out of cash and missing the right competences are the second and third reasons for startup failures[2].

The risk described above comes from a disruption between the two levels of application expression, the feature level and the execution level. To avoid this risk and allow a continuous development process, we propose a tool to automatically map one level onto the other, and make the transition.

We focus on web applications driven by users requests and developed in Javascript using the *Node.js*[3] execution environment. Javascript is widely adopted[4][5] to develop web applications, and its event-loop model is very similar to a pipeline architecture. So we propose a compiler to transform an application into a pipeline of parallel stages communicating by message streams. We named these stages *fluxions*, by contraction between a flux and a function.

We present a proof of concept for this compilation approach. Section 2 describes the execution environment targeted by this compiler. Then, section 3 presents the compiler, and section 4 its evaluation. Section 5 compare our work with related works. And finally, we conclude this paper.

[2]https://www.cbinsights.com/blog/startup-failure-post-mortem/
[3]https://nodejs.org/
[4]http://githut.info/
[5]http://stackoverflow.com/tags

2. FLUXIONAL EXECUTION MODEL

This section presents an execution model to provide scalability to web applications with a granularity of parallelism at the function level. Functions are encapsulated in autonomous execution containers with their state, so as to be mobile and parallel, similarly to the actors model. The communications are similar to the dataflow programming model, which allows to reason on the throughput of these streams, and to react to load increases [4].

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⟩. The following paragraphs present the *fluxions* and the messaging system to carry the communications between *fluxions*, and then an example application using this execution model.

2.1 Fluxions and Messaging System

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 to downstream *fluxions* on its output streams ⟨streams⟩. The *context* stores the state on which a *fluxion* relies between two message receptions. The messaging system queues the output messages for the event loop to process them later by calling the downstream *fluxions*.

In addition to message passing, the execution model allows *fluxions* to communicate by sharing state between their *contexts*. The fluxions that need this synchronization are grouped with the same tag, and loose their independence.

There are two types of streams, *start* and *post*, which correspond to the nature of the rupture point producing the stream. A variable created within a chain of *post* streams requires more synchronization than a variable created upstream a *start* stream. The two types and implications of rupture points are further detailed in section 3. *Start* rupture points are indicated with a double arrow ($\twoheadrightarrow$ or `>>`) and *post* rupture points with a simple arrow ($\rightarrow$ or `->`).

⟨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⟩	⊨	*Source language with* ⟨stream⟩ *placeholders*

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

2.2 Example

```
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

The fluxional execution model is illustrated with an example application presented in listing 1. This application reads a file, and sends it back along with a request counter. The `handler` function, line 5 to 10, receives the input stream of requests. The `count` variable at line 3 counts the requests, and needs to be saved between two messages receptions. The `template` function formats the output stream to be sent back to the client. The `app.get` and `res.send` functions, lines 5 and 8, interface the application with the clients. Between these two interface functions is a chain of three functions to process the client requests : `app.get` $\twoheadrightarrow$ `handler` $\rightarrow$ `reply`. This chain of functions is transformed into a pipeline, expressed in the high-level fluxionnal language in listing 2. The transformation process between the source and the fluxional code is explained in section 3.

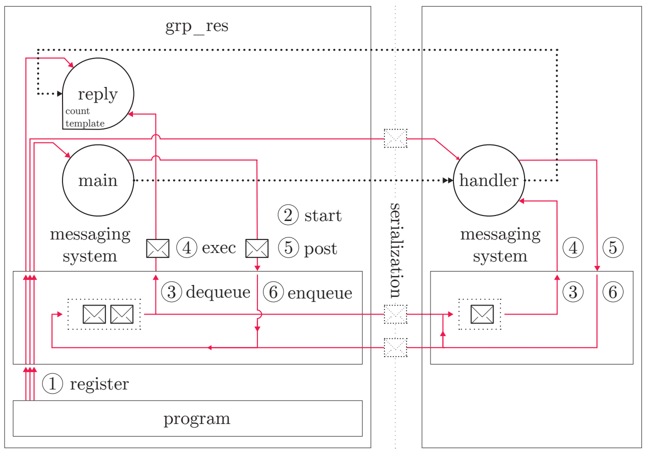


Figure 2: The fluxionnal execution model in details

The execution is illustrated in figure 2. The dashed arrows between fluxions represent the message streams as seen in the fluxionnal application. The plain arrows represent the operations of the messaging system during the execution. These steps are indicated by numeroted circles. The *program* registers its fluxions in the messageing system, ①. The fluxion *reply* has a context containing the variable `count` and `template`. When the application receives a request, the first fluxion in the stream, *main*, queues a `start` message containing the request, ②. This first message is to be received by the next fluxion *handler*, ③, and triggers its execution, 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.

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

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

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

flx reply & grp_res {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

The chain of functions from listing 1 is expressed in the fluxional language in listing 2. The fluxion `handler` doesn't have any dependencies, so it can be executed in a parallel event-loop. The fluxions `main` and `reply` belong to the group `grp_res`, indicating their dependency over the variable `res`. The group name is arbitrarily chosen by the compiler. All the fluxions inside a group are executed sequentially on the same event-loop, to protect against concurrent accesses.

The variable `res` is created and consumed within a chain of *post* stream. Therefore, it is exclusive to one request and cannot be propagated to another request. It doesn't prevent the whole group from being replicated. However, the fluxion `reply` depends on the variable `count` created upstream the *start* stream, which prevents this replication. If it did not rely on this state, the group `grp_res` would be stateless, and could be replicated to cope with the incoming traffic.

This execution model allows to parallelize the execution of an application as a pipeline, as with the fluxion `handler`. And some parts are replicated, as could be the group `grp_res`. 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 high-level language but to automate the architectural shift. We present the compiler to automate this architectural shift in the next section.

3. FLUXIONNAL COMPILER

The source languages we focus on should offer higher-order functions and be implemented as an event-loop with a global memory. Javascript is such a language and is often implemented on top of an event-loop, like in *Node.js*. We developed a compiler that transforms a *Node.js* application into a fluxional application compliant with the execution model described in section 2. Our compiler uses the *estools*[6] suite to parse, manipulate and generate source code from Abstract Syntax Tree (AST). And it is tailored for – but not limited to – web applications using *Express*[7], the most used *Node.js* web framework.

[6]https://github.com/estools
[7]http://expressjs.com/

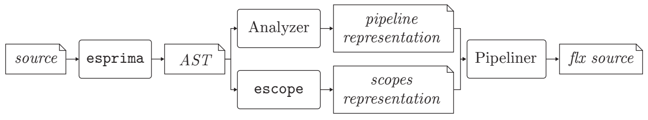


Figure 3: Compilation chain

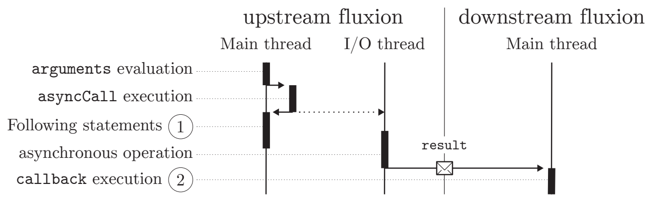
The chain of compilation is described in figure 3. The compiler extracts an AST from the source with `esprima`. From this AST, the *Analyzer* step identifies the limits of the different application parts and how they relate to form a pipeline. This first step outputs a pipeline representation of the 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. Application parts are the stages, and rupture points are 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 indicate rupture points between two application parts. Figure 4 shows a code example of a rupture point with the illustration of 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.



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

Figure 4: Rupture point interface

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. There are three kinds of callbacks, but only two are asynchronous: listeners and continuations. The two corresponding types of rupture points are *start* and *post*.

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

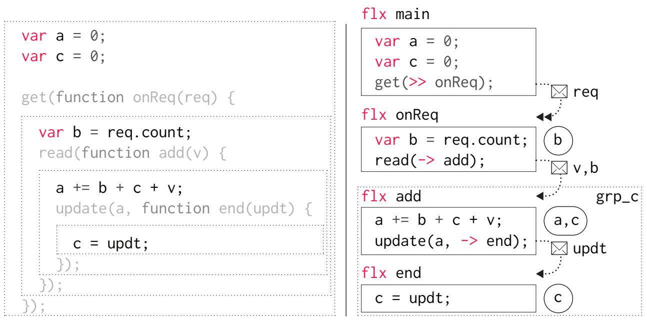
3.1.2 Detection

The compiler uses a list of common asynchronous callees, like the `express` and file system methods. This list can be augmented to match asynchronous callees 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 tests if one of the arguments is of type `function`. Some callback functions are declared *in situ*, and are trivially detected. For variable identifiers, and other expressions, the analyzer tries to detect their type. The analyzer walks back the AST to track their assignations and modifications, so as to 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 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, the compiler can replace the broken closures in three different ways. We present these three alternatives in figure 5.



```
var a = 0;
var c = 0;

get(function onReq(req) {
  var b = req.count;
  read(function add(v) {
    a += b + c + v;
    update(a, function end(updt) {
      c = updt;
    });
  });
});
```

```
flx main
var a = 0;
var c = 0;
get(>> onReq);

flx onReq
var b = req.count;
read(-> add);

flx add
a += b + c + v;
update(a, -> end);

flx end
c = updt;
```

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 to the context of its fluxion.

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

Stream.

If a modified variable is read by downstream application parts, then the pipeliner makes the upstream 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 causing inconsistencies. 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 step 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, like `b`, 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 reading of the variable by the upstream fluxion happens after the modification by the downstream fluxion.

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. To respect the semantics of the source application, we cannot tolerate inconsistencies. Therefore, the pipeliner groups all the fluxions sharing this variable with the 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 step groups the fluxion `add` and `end` with the tag `grp_c` to allow the two fluxions to share this variable.

4. REAL TEST CASE

This section presents a test of the compiler on a real application, gifsockets-server[8]. This test proves the possibility for an application to be compiled into a network of independent parts. It shows the current limitations of this isolation and the modifications needed on the application to circumvent them. This section then presents future works.

```
var express = require('express'),
    app = express(),
    routes = require('gifsockets-middleware'),
    getRawBody = require('raw-body');

function bodyParser(limit) {
  return function saveBody(req, res, next) {
    getRawBody(req, {
      expected: req.headers['content-length'],
      limit: limit
    }, function (err, buffer) {
      req.body = buffer;
      next();
    });
  };
}

app.post('/image/text', bodyParser(1 * 1024 * 1024),
     routes.writeTextToImages);
app.listen(8000);
```

Listing 3: Simplified version of gifsockets-server

[8]https://github.com/twolfson/gifsockets-server

This application, simplified in listing 3, is a real-time chat using gif-based communication channels. It was selected in a previous work [5] from the `npm` registry because it depends on `express`, it is tested, working, and simple enough to illustrate this evaluation. The server transforms the received text into a gif frame, and pushes it back to a never-ending gif to be displayed on the client.

On line 18, 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 processing the request with the next function, `routes.writeTextToImages`.

4.1 Compilation

We compile this application with the compiler detailed in section 3. Listing 4 presents the compilation result. The function call `app.post`, line 18, is a rupture point. However, its callbacks, `bodyParser` and `routes.writeTextToImages` are evaluated as functions only at runtime. For this reason, the compiler ignores this rupture point, to avoid interfering with the evaluation.

```
1  flx main & express {req}
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

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 appended 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, as if they belong to the same group, 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.

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 prevents 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 parallelized.

4.2.1 Variable `req`

The variable `req` is read in fluxion `main`, lines 10 and 11. Then its property `body` is associated to `buffer` in fluxion `anonymous_1000`, 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 the next callback through the function `next`. We explain further modification of this function in the next paragraph.

4.2.2 Closure `next`

The function `next` is a closure provided by the `express Router` to continue the execution with the next function to handle the client request. Because it indirectly relies on the variable `req`, 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 modifications below.

```
1  flx anonymous_1000
2  -> express_dispatcher
3    function (err, buffer) {
4      req.body = buffer;
5      next_placeholder(req, -> express_dispatcher);
6    }
7
8  flx express_dispatcher & express {req}
9  -> null
10   function (modified_req) {
11     merge(req, modified_req);
12     next();
13   }
```

Listing 5: Simplified modification on the compiled result

In listing 3, the function `next` is a continuation allowing the anonymous callback, line 11, to call the next function to handle the request. To isolate the anonymous callback into `anonymous_1000`, `next` is replaced by a rupture point. This replacement is illustrated in listing 5. The `express Router` registers a fluxion named `express_dispatcher`, line 8, to continue the execution after the fluxion `anonymous_1000`. This fluxion is in the same group `express` as the `main` fluxion, hence it has access to the original variable `req`, and to the original function `next`. The call to the original `next` function is replaced by a placeholder to push the stream to the fluxion `express_dispatcher`, line 5. The fluxion `express_dispatcher` receives the stream from the upstream fluxion `anonymous_1000`, merges back the modification in the variable `req` to propagate the side effects, and finally calls the original function `next` to continue the execution, line 12.

After the modifications detailed above, the server works as expected. 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 implement the compilation process presented into the runtime. A just-in-time compiler would allow to identify callbacks dynamically evaluated, and to analyze the memory to identify side-effects propagations instead of relying only on the source code. Moreover, this memory analysis would allow the closure serialization required to compile application using higher-order functions.

5. RELATED WORKS

Splitting a task into independent parts goes back to the Actor's model, functional programming [12] and the following works on DataFlow leading to Flow-based Programming (FBP) and Functional Reactive programming (FRP) [8]. Both FBP and FRP, recently got some attention in the Javascript community with *NoFlo*[9], *Bacon.js*[10] and *react*[11].

The execution model we presented in section 2, is inspired by works on scalability for very large systems, like the Staged Event-Driven Architecture (SEDA) by Matt Welsh [20] and by the MapReduce architecture [7]. It also drew its inspiration from more recent work following SEDA like Spark [21], MillWheel [1], Naiad [15] and Storm [19]. The first part of our work stands upon these thorough studies. However, we believe that it is difficult for most developers to distribute the state of an application. This belief motivated us to propose a compiler from an imperative 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.* [9]. However, as in similar works [17], it requires annotations from developers, therefore partially conserves the disruption with the feature-based development. Our approach discards the need for annotations, thus targets a broader range of developers than only ones experienced with parallel development.

A great body of work focuses on parallelizing sequential programs [3, 13, 14, 18]. Because of the synchronous execution of a sequential program, the speedup of parallelization is inherently limited [2, 11]. On the other hand, our approach is based on an asynchronous programming model. Hence the attainable speedup is not limited by the main synchronous thread of execution.

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. We presented an example of a compiled application to show the limits of this approach. The parallelism of this approach allows code-mobility which may lead to a better scalability. We believe it can enable the scalability required by highly concurrent web applications without discarding the familiar, feature-driven programming models.

[9]http://noflojs.org/

[10]https://baconjs.github.io/

[11]https://facebook.github.io/react/

References

[1] T Akidau and A Balikov. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *VLDB* (2013).

[2] G Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: (1967).

[3] U Banerjee. *Loop parallelization*. 2013.

[4] T Bartenstein and Y Liu. "Rate Types for Stream Programs". In: *OOPSLA* (2014).

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

[6] A Clements, M Kaashoek, N Zeldovich, R Morris, and E Kohler. "The scalable commutativity rule". In: *SOSP* (2013).

[7] J Dean and S Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI* (2004).

[8] C Elliott and P Hudak. "Functional reactive animation". In: *SIGPLAN* (1997).

[9] R Fernandez, M Migliavacca, E Kalyvianaki, and P Pietzuch. "Making state explicit for imperative big data processing". In: *USENIX ATC* (2014).

[10] A Fox, S Gribble, Y Chawathe, E Brewer, and P Gauthier. "Cluster-based scalable network services". In: *SOSP* (1997).

[11] N Gunther. "A general theory of computational scalability based on rational functions". In: *ArXiv e-prints* (2008).

[12] J Hughes. "Why functional programming matters". In: *The computer journal* (1989).

[13] F Li, A Pop, and A Cohen. "Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs". In: *IEEE Micro* (2012).

[14] N Matsakis. "Parallel Closures A new twist on an old idea". In: *HotPar* (2012).

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

[16] D Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* (1972).

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

[18] C Radoi, S Fink, R Rabbah, and M Sridharan. "Translating imperative code to MapReduce". In: *OOPSLA* (2014).

[19] A Toshniwal, J Donham, N Bhagat, S Mittal, D Ryaboy, S Taneja, A Shukla, K Ramasamy, J Patel, S Kulkarni, J Jackson, K Gade, and M Fu. "Storm@ twitter". In: *SIGMOD* (2014).

[20] M Welsh, S Gribble, E Brewer, and D Culler. "A design framework for highly concurrent systems". In: *University of California, Berkeley* (2000).

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