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-oriented approach allowing to quickly react to users feedbacks. However, this approach poorly scales in performance. Yet, the audience 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 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 (flux in french) 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.

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. INTRODUCTION

"Release early, release often", *"Fail fast"*. The growth of real-time web services 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 need is the first reason for startup failure[1]. 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 audience. However, feature-driven development best practices are hardly compatible with the required parallelism. The features are organized in modules which overlap and disturb the organization of a parallel execution [6, 13, 17]. 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, 21, 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 as the second and third reasons for startup failures are running out of cash, and missing the right competences[1].

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 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, developed in Javascript using the *Node.js*[2] execution environment. We have no doubt on Javascript adoption[3][4], and strive for performance. 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 messages streams. We named these stages *fluxions*, by contraction between a flux and a function.

We present a tool as 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.

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

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 messaging system queues the output messages for the event loop to later process them by calling the downstream *fluxions*. The *context* stores 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 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 yielding the stream. We differentiate the two types with two different arrows, double arrow (↠ or >>) for *start* rupture points and simple arrow (→ or ->) for *post* rupture points. The two types of rupture points are further detailed in section 3.1.1.

⟨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 with inline* ⟨stream⟩

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

2.2 Example

Listing 1 presents a simple web application to illustrate the fluxional execution model. This application reads a file, and sends it back along with a request counter. The equivalent fluxional code is in listing 2. The transformation pocess is explained later, in section 3. The execution cycle of the example is illustrated in figure 2.

Bigger circles represent registered fluxions, while smaller ones indicate the step in the execution. The fluxion *reply* has a context containing the variable `count` and `template`. 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. When the application receives a request, this fluxion triggers the flow with a `start` message containing the request, 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.

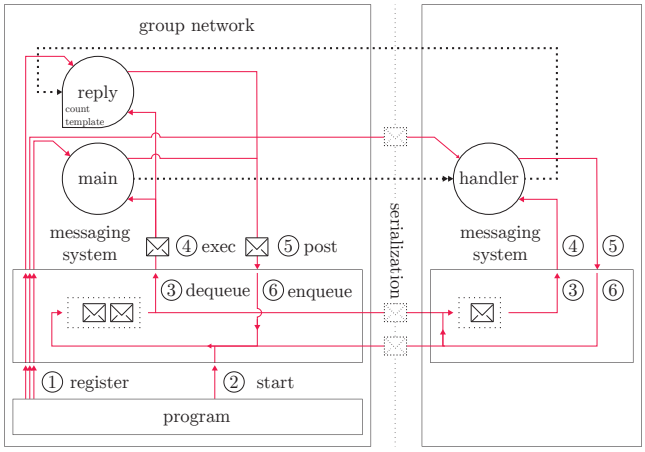


Figure 2: The fluxionnal execution model in details

```
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 `handler` function, line 5 to 10, receives the input stream of request. The `count` variable at line 3 counts the requests. This variable needs to be persisted in the fluxion *context*. The `template` function formats the output stream to be sent back to the client. The `app.get` and `res.send` functions, respectively line 5 and 8, interface the application with the clients. And between these two interface functions is a chain of three functions to process the client requests : `app.get` ↠ `handler` → `reply`. This application is transformed into the high-level fluxionnal language in listing 2

which is illustred in Figure 2.

```
flx main & network
>> 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 & 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

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 goes through the fluxion `reply` to be sent 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` doesn't have any dependencies, hence it can be executed in parallel, and replicated.

The last fluxion, `reply`, depends on the variable `res` created by the first fluxion, `main`. This variable is carried by the stream through the chain of fluxion to the fluxion `reply` that depends on it. The group `network` depends on this variable because it holds the references to the network sockets. This variable is created and consumed inside this group, therefore, it doesn't prevent the whole group from being replicated.

However, the last fluxion, `reply`, depends on its context to hold the variable `count` which prevents this replication. If it did not rely on this state, the group `network` would be stateless, and could be replicated to cope with the incoming traffic.

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, as 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.

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, but it is often implemented on top of an event-loop. *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.

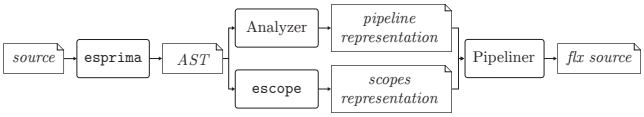


Figure 3: Compilation chain

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) 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.

Our compiler uses the *estools*[5] suite to parse, manipulate and generate source code from Abstract Syntax Tree (AST). Our implementation is based on the work by Ryan Dahl : *Node.js*, as well as on one of the best-known *Node.js* web framework : *Express*[6].

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.

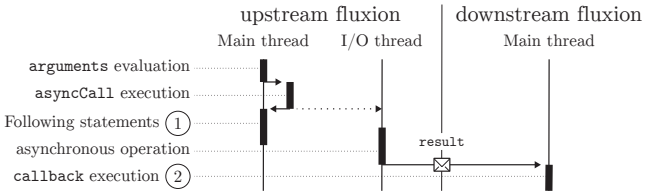
3.1.1 Rupture points

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 indicate such rupture point 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.

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 are 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

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




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

Figure 4: Rupture point interface

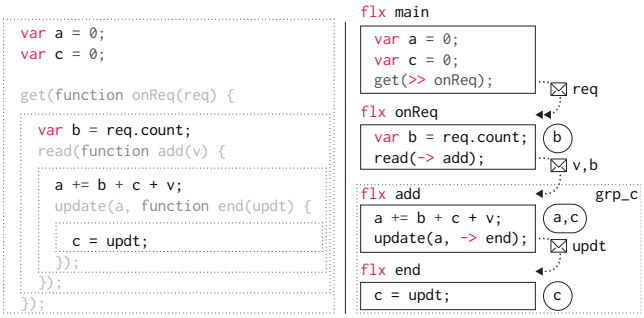


Figure 5: Variable management from Javascript to the high-level fluxionnal language

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 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 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 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 with the example figure 5.

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 variable is modified inside an application part, and read inside 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 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 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 within 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 CASE TEST

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

We present a test of our compiler on a real application, gifsockets-server[7]. This application was selected from the `npm` registry because it depends on `express`, it is tested, working, and simple enough to illustrate this evaluation. It is part of the selection from a previous work [5].

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

This application is a real-time chat using gif-based communication channels. 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. Listing 3 is a simplified version of this application.

```
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       req.body = buffer;
13       next();
14     });
15   };
16 }
17
18 app.post('/image/text', bodyParser(1 * 1024 * 1024),
       routes.writeTextToImages);
19 app.listen(8000);
```

Listing 3: Simplified version of gifsockets-server

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. 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.

The compilation result is in listing **??**. 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 **??**, the fluxion `anonymous_1000` modifies the object `req`, line **??**, to store the text of the received request, and it calls `next` to continue the execution, line **??**. 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. This test highlights the current limitations of the compiler, and presents future works to circumvent them.

4.2.1 *Variable* `req`

The variable `req` is read in fluxion `main`, lines **??** and **??**. Then its property `body` is associated to `buffer` in fluxion `anonymous_1000`, line **??**. 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 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.

```
1  flx main & express
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     next_placeholder(req, -> express_dispatcher);
25   }
26
27 flx express_dispatcher & express
28 -> null
29   merge(req, msg.req);
30   next();
```

Listing 4: Simplified modification on the compiled result

Originally, the function `next` is the continuation to allow the anonymous callback on line 11, 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 result of this replacement is illustrated in listing 4. The `express Router` registers a fluxion named `express_dispatcher`, line 27, 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 network sockets, to the original variable `req`, and to the original function `next`. The call to the original `next` function in the anonymous callback is replaced by a placeholder to push the stream to the fluxion `express_dispatcher`, line 24. 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, before calling the original function `next` to continue the execution, line 30.

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 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

The idea to split a task into independent parts goes back to the Actor's model [12], functional programming [13] and the following works on DataFlow leading up to Flow-based Programming (FBP) and Functional Reactive programming (FRP) [8]. Both FBP and FRP, recently got some attention in the Javascript community with the projects *NoFlo*[8], *Bacon.js*[9] and *react*[10].

The execution model we presented in section 2, is inspired by some works on scalability for very large systems, like the Staged Event-Driven Architecture (SEDA) by Matt Welsh [21] and later by the MapReduce architecture [7]. It also drew its inspiration from more recent work following SEDA. Among the best-known following works, we cited in the introduction Spark [22], MillWheel [1], Naiad [16] and Storm [20]. 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 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.* [9]. However, as in similar works [18], it requires annotations from developers, therefore partially conserves the disruption with the feature-based development. Our approach avoids the need for annotations, thus targets a broader range of developers, and not only ones experienced with parallel development.

A great body of work focus on parallelizing sequential programs [3, 14, 15, 19]. Because of the synchronous execution of a sequential program, the speedup of parallelization is inherently limited. 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 [2, 11].

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-based programming models.

[8] `http://noflojs.org/`

[9] `https://baconjs.github.io/`

[10] `https://facebook.github.io/react/`

References

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

[2] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *AFIPS Spring Joint Computer Conference, 1967. AFIPS '67 (Spring). Proceedings of the.* Vol. 30. 1967, pp. 483–485. DOI: doi:10.1145/1465482.1465560.

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

[4] Thomas W Bartenstein and Yu David Liu. "Rate Types for Stream Programs". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2014), pp. 213–232. DOI: 10.1145/2660193.2660225.

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

[6] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. "The scalable commutativity rule". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13.* New York, New York, USA: ACM Press, Nov. 2013, pp. 1–17. DOI: 10.1145/2517349.2522712.

[7] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proc. of the OSDI - Symp. on Operating Systems Design and Implementation.* Vol. 51. 1. 2004, pp. 137–149. DOI: 10.1145/1327452.1327492. arXiv: 10.1.1.163.5292.

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

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

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

[11] NJ Gunther. "A general theory of computational scalability based on rational functions". In: *arXiv preprint arXiv:0808.1431* (2008).

[12] C Hewitt, P Bishop, and R Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence* (1973).

[13] John Hughes. “Why functional programming matters”. In: *The computer journal* 32.April 1989 (1989), pp. 1–23. DOI: 10.1093/comjnl/32.2.98.

[14] Feng Li, Antoniu Pop, and Albert Cohen. “Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs”. English. In: *IEEE Micro* 32.4 (July 2012), pp. 19–31. DOI: 10.1109/MM.2012.49.

[15] Nicholas D Matsakis. “Parallel Closures A new twist on an old idea”. In: *HotPar'12 Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012), pp. 5–5.

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

[17] DL Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Communications of the ACM* (1972).

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

[19] C Radoi, SJ Fink, R Rabbah, and M Sridharan. “Translating imperative code to MapReduce”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2014).

[20] Ankit Toshniwal, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, and Maosong Fu. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. New York, New York, USA: ACM Press, June 2014, pp. 147–156. DOI: 10.1145/2588555.2595641.

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

[22] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion 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), pp. 10–10.