Automatic pipeline parallelism for Javascript : Abstract scalability constraints from the developer

Etienne Brodu
etienne.brodu@insa-lyon.fr
IXXI – ENS Lyon
15 parvis René Descartes – BP 7000
69342 Lyon Cedex 07 FRANCE

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 distributed approach. 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 may allow later to provide code mobility to dynamically cope with audience growth and decrease.

We want to propose a compiler to transform a Javascript, monolithic, web application into a network of small independent parts communicating by message streams. We named these parts *fluxions*, by contraction between a flux and a function. We expect 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 products (MVP). In a matter of hours, it is possible to upload 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 validate quickly 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 complies successfully 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[9]. MapReduce [5] and the Staged Event-driven Architecture (SEDA) [24] are famous examples of that trend, using a pipeline architecture. 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 [26], MillWheel [1], Timestream [21], Naiad [18] and Storm [17][22], and many, 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 economic evolution of the project. The number two and three reasons for startup failures are running out of cash, and not having the right competences in the team.

To lift the risks described above, we propose a tool to compile the initial code base into a high-level language compatible with the more efficient processing model. We focus on web applications driven by users requests, developed in

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

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 the 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.

This short paper presents 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 **??**. We compare our work with related works in section 5. And finally, we conclude this paper.

2. FLUXIONAL EXECUTION MODEL

Many frameworks for distributed systems are renowned for their performances[1, 14, 17, 24, 25, 26]. However, we focus on a compilation approach to replace the shift in programming model rather than the performance of the runtime. We present in this section an extremely simplified but generic execution model inspired by the literature, only to support the confirmation of feasibility for the compilation process detailed in section 3. The execution model is not distributed on remote machines, however it isolates the execution of fluxions in different processes to reproduce the execution conditions of a distributed execution model. We are interested in the problems arising from this isolation.

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 rely 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 impossible 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 and atomicity of operations of each fluxion on the heap. The following sentence is poorly constructed This organization shows that the more the memory is shared, the harder it is to distribute fluxions on different workers to allow parallelisation of their execution.

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

⟨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*

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.

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, but each worker still has its own message queue.

The messaging system sends messages to the isolated worker hosting the destination fluxion. The worker containing the messaging system locally hosts fluxions that need access to the external network. The life cycle of a fluxional application is illustrated in figure 1. Circles represent registered fluxions. The fluxion *reply* as a context containing the variable `count`. The red arrows represent the actual message path 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 using the function `register(<name>, <fn>, <context>)`, ①.

When a new REST request `GET` 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 empty. This cycle starts again for each new incoming request causing a *start* message.

Algorithms 1 and 2 describe the behavior of the messaging

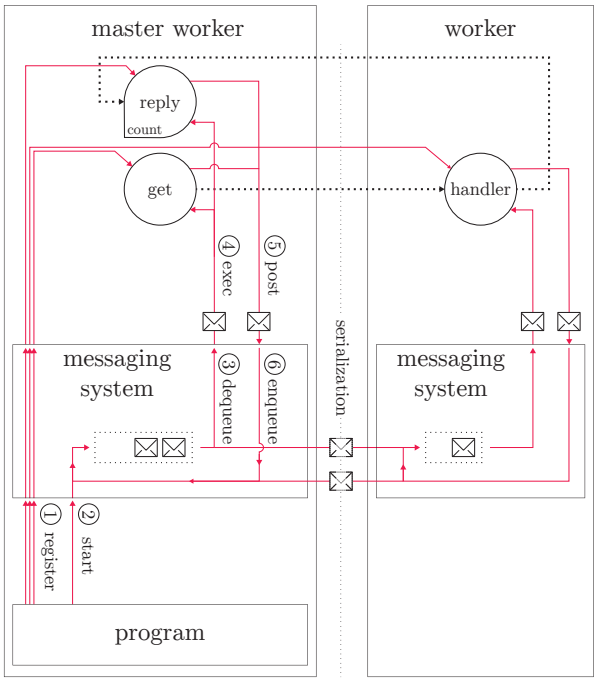

Figure 1: The fluxionnal execution model in details

system after the `start` function invocation.

Algorithm 1 Message queue walking algorithm

```
function LOOPMESSAGE()
    while msg presents in msgQueue do
        msg ← DEQUEUE()                    ▷ ③
        PROCESSMSG(msg)
    end while
end function
```

Algorithm 2 Message processing algorithm

```
function PROCESSMSG(msg)
    for dest in msg.dest do
        worker ← lookup(dest)
        WORKER.SEND(fluxion, msg.body)     ▷ ④
    end for
end function
```

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[3], and in listing 1. In this source code, some points are worth noticing.

— The `handler` function, line 5 to 11, contains the logic we want to split into the fluxional processing chain. It receives the user request in the variable `res` which is used by the last function of the chain, `reply`.

— The `count` object at line 3 is a persistent memory that increments the request counter. This object needs to be mapped to a fluxion *execution context* in the fluxional execution model.

— 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      var code = ('' + data).replace(/\n/g, '<br>').replace
           (/ /g, ' ');
9      res.send(err || 'downloaded ' + count + ' times<br><
           br><code>' + code + '</code>');
10   });
11 });
12
13 app.listen(8080);
14 console.log('>> listening 8080');
```

Listing 1: Simple web application. this application replies to every user request with its own source code and the value of a request counter

This application is transformed manually into the fluxions chain depicted 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    console.log('>> listening 8080');
10
11 flx handler
12 -> reply [res]
13   function handler(req, res) {
14     fs.readFile(__filename, -> reply);
15   }
16
17 flx reply {count}
18 -> null
19   function reply(error, data) {
20     count += 1;
21     var code = ('' + data).replace(/\n/g, '<br>').replace
           (/ /g, ' ');
22     res.send('downloaded ' + count + ' times<br><br><code
           >' + code + '</code>' );
23   }
```

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 14.

— The `reply` fluxion increments the counter, line 20, formats the reply, and sends it back to the user using the function `res.send`, line 22.

TODO transition missing 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.

3. FLUXIONNAL COMPILER

Web applications are currently, mostly written in Java. The langage proposes both data encapsulation in object and a threading model that allow the development of parallel applications. But, this approach is error-prone, and leads to deadlocks and other synchronization problems [**Adya2002**]. Since 2009, *Node.js*[4] propose 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. We develop a compiler that transforms a *Node.js* application into a fluxional system compliant with the architecture described in section 2. Our compiler uses a new approach to find independent parts in *Node.js* applications. It finds rupture points that represent limitations between potential 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.

Section 3.1 define rupture points, and explains how the compiler detects them. Section 3.2 explains how the compiler distribute the central memory into isolated fluxions.

3.1 Analyzer

3.1.1 Rupture points

A rupture point is a call of a loosely coupled function. It is an asynchronous call without subsequent synchronization with the caller. In *Node.js*, I/O operations are asynchronous functions. That is a function call that resumes immediately, with a function to process later the result of the operation : the callback. The callback is loosely coupled with the initial caller, because they are not executed on the same call stack. This rupture in the call stack marks out the separation between two independent application parts. In the *Node.js* environment, these two application parts are executed sequentially on the same thread to avoid concurrent memory accesses. But the compiler isolates their memories to reveal a pipeline parallelism, as defined in[11].

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 callback, 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.

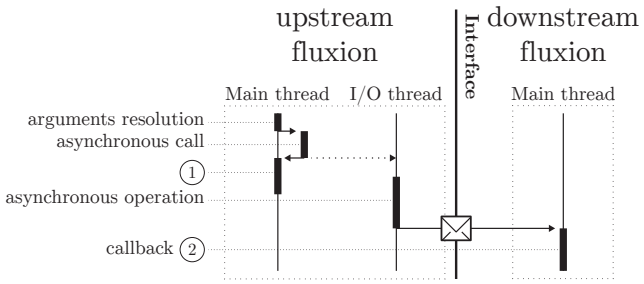

Figure 2: Basic rupture point interface. The rupture point interface is placed between the asynchronous operation and the callback in between the two call stacks.

3.1.2 Detection

Listeners and continuations are not asynchronous because they are different than any other function, but because they are called asynchronously. 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, or 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. Ideally, our tracking method would be a simplified version of the points-to analysis [23].

3.2 Linker

In an stream processing, there is roughly two kinds of usage of the global memory : data and state [8]. Naively, the data represent a communication channel between different point in the application space, and the state represent a communication channel between different instant in time. Data flows from stage to stage through the pipeline, and is never stored on any fluxion. In the source application, it is stored in the heap, only as a buffer between the different callbacks. 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 - the *context* - is not chained to its parent. As a rupture points is always between a child scope and its parent, it eventually breaks a chain of scopes, and makes the child unable to access its parent as expected. The parent is in the upstream fluxion, and the child in the downstream fluxion. This situation, if not resolved, introduces 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 only the 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-effect between the two scopes. We call these side-effects scope leaking. A very simple example of scope leaking is illustrated in listing 3. The scope analysis previously presented is unable to take scope leaking into account. Our compiler is currently in early stage of development. It is unable to analyze the memory deeply enough to provide a sound and complete analysis. Therefore it might lead to runtime errors.

```
1 function scopeLeak(a)
2 {
3   a.item = "changed";
4 }
5
6 var b = {item: "unchanged"};
7
8 scopeLeak(b);
9 console.log(b); // "changed";
```

Listing 3: Example of a simple scope leak

3.2.2 *State sharing*

Depending on the result of the previous analysis, 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.

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 [10, 12], and the BASE semantics[9]. 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 with other fluxions, either in its context, or streams, they need to be hosted on the same worker.

4. EVALUATION

We tested our compiler on two applications, and present here the results.

TODO continue

5. RELATED WORKS

The execution model, is inspired by some works on scalability for very large system, like the Staged Event-Driven Architecture (SEDA) of Matt Welsh[24], System S developped in the IBM T. J. Watson research center[14, 25], and later the MapReduce architecture[5]. It also drew its inspiration from more recent work following SEDA. Among the best-known following works, we cited in the introduction Spark [26, 27], MillWheel [1], Timestream [21] and Storm [17]. The idea to split a task into independent parts goes back to the Actor's model[13] in 1973, and to Functional programming, like Lucid[2] in 1977 and all the following works on DataFlow leading up to Flow-Based programming (FBP)[**Morrison1994a**] and Functional Reactive Programming (FRP)[6]. Both FBP and FRP, recently got some attention in the Javascript community with, respectively, the projects *NoFlo*[19] and *Bacon.js*[20].

The first part of our work stands upon these thorough studies, however, we are taking a new approach on the second part of our work, to transform the sequential programming paradigm into a network of communicating parts known to have scalability advantages. Promises[16] are related to our work as they are abstractions from a concurrent programming style, to an asynchronous and parallel execution model.

However, our approach using Node.js callback asynchronism to automate this abstraction seems unexplored yet.

The compiler uses AST modification, as described in[15]. Our implementation is based on the work by Ryan Dahl : *Node.js*[4], as well as on one of the best-known web framework available for *Node.js* : *Express*[7].

TODO talk about that : [8], and similar
TODO talk about static analysis

6. CONCLUSION

In this paper, we presented our work on a high-level language allowing a high-level code reasoning. We presented a compiler to transform a *Node.js* web application into a network of independent parts communicating by message streams. To identify these parts, the compiler spots rupture points in the application indicating an independence between two parts, possibly leading to parallelism and memory distribution. We also presented the execution model to operate an application expressed in our high-level language. This distributed approach allows code-mobility which may lead to a better scalability. We believe this high-level approach can enable the scalability required by highly concurrent web applications without discarding the familiar monolithic and asynchronous programming model used in *Node.js*.

Références

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

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

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

[4] Ryan Dahl. *Node.js*. 2009.

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

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

[7] *Express*.

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

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

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

[11] MI Gordon, W Thies et S Amarasinghe. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs". In : *ACM SIGOPS Operating Systems ...* (2006).

[12] Coda Hale. *You Can't Sacrifice Partition Tolerance | codahale.com*. 2010.

[13] 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).

[14] 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).

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

[16] B Liskov et L Shrira. *Promises : linguistic support for efficient asynchronous procedure calls in distributed systems*. 1988.

[17] Nathan Marz, James Xu, Jason Jackson et Andy Feng. *Storm*. 2011.

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

[19] *NoFlo*.

[20] Juha Paananen. *Bacon.js*. 2012.

[21] 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).

[22] A Toshniwal et S Taneja. "Storm@ twitter". In : *Proceedings of the ...* (2014).

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

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

[25] 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).

[26] 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).

[27] 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).