Liquid IT : Toward a better compromise between development scalability and performance scalability not definitive

Etienne Brodu

December 11, 2015

Abstract

TODO translate from below when ready

Résumé

Internet étend nos moyens de communications, et réduit leur latence ce qui permet de développer l'économie à l'échelle planétaire. Il permet à chacun de mettre un service à disposition de milliards d'utilisateurs, en seulement quelques heures. La plupart des grands services actuels ont commencé comme de simples applications créées dans un garage par une poignée de personnes. C'est cette facilité à l'entrée qui a permis jusqu'à maintenant une telle croissance sur le web. Google, Facebook ou Twitter en sont quelques exemples. Au cours du développement d'une application, il est important de suivre cette croissance, au risque de se faire rattraper par la concurrence. Le développement est guidé par les besoins en terme de fonctionnalités, afin de vérifier rapidement si le service peut satisfaire l'audience. On parle d'approche modulaire des fonctionnalités. Des langages tel que Ruby ou Java se sont imposés comme les langages du web, justement parce qu'ils suivent cette approche qui permet d'intégrer facilement de nouvelles fonctionnalités.

Si une application répond correctement aux besoins, elle atteindra de manière virale un nombre important d'utilisateurs. Son audience peut prendre plusieurs ordres de grandeurs en quelques jours seulement, ou même en quelques heures suivant comment elle est relayée. Une application est dite *scalable* si elle peut absorber ces augmentations d'audience. Or il est difficile pour une application suivant l'approche modulaire d'être *scalable*.

Au moment où l'audience commence à devenir trop importante, il est nécessaire de modifier l'approche de développement de l'application. Le plus souvent cela implique de la réécrire complètement en utilisant des infrastructures *scalables* qui imposent des modèles de programmation et des API spécifiques, qui représentent une charge de travail conséquente et incertaine. De plus, l'équipe de développement dois concilier cette nouvelle approche de développement *scalable*, avec la demande en fonctionnalités. Aucun langage n'a clairement réussi le compromis entre ces deux objectifs.

Pour ces raisons, ce changement est un risque pour la pérennité de l'application. D'autant plus que le cadre économique accorde peu de marges d'erreurs, comme c'est le cas dans la plupart des start-up, mais également dans de plus grandes structures.

Cette thèse est source de propositions pour écarter ce risque. Elle repose sur les deux observations suivantes. D'une part, Javascript est un langage qui a gagné en popularité ces dernières années. Il est omniprésent sur les clients, et commence à s'imposer également sur les serveurs avec Node.js. Il a accumulé une communauté de

développeurs importante, et constitue l'environnement d'exécution le plus largement déployé. De ce fait, il se place maintenant de plus en plus comme le langage principal du web, détrônant Ruby ou Java. D'autre part, l'exécution de Javascript s'assimile à un pipeline. La boucle événementielle de Javascript exécute une suite de fonctions dont l'exécution est indépendante, mais qui s'exécutent sur un seul cœur pour profiter d'une mémoire globale.

L'objectif de cette thèse est de maintenir une double représentation d'un code Javascript grâce à une équivalence entre l'approche modulaire, et l'approche pipeline d'un même programme. La première répondant aux besoins en fonctionnalités, et favorisant les bonnes pratiques de développement pour une meilleure maintenabilité. La seconde proposant une exécution plus efficace que la première en permettant de rendre certaines parties du code relocalisables en cours d'exécution.

Nous étudions la possibilité pour cette équivalence de transformer un code d'une approche vers l'autre. Grâce à cette transition, l'équipe de développement peut continuellement itérer le développement de l'application en suivant les deux approches à la fois, sans être cloisonné dans une, et coupé de l'autre.

Nous construisons un compilateur permettant d'identifier les fonctions de Javascript et de les isoler dans ce que nous appelons des Fluxions, contraction entre fonctions et flux. Un conteneur qui peut exécuter une fonction à la réception d'un message, et envoyer des messages pour continuer le flux vers d'autres fluxions. Les fluxions sont indépendantes, elles peuvent être déplacées d'une machine à l'autre.

Nous montrons qu'il existe une correspondance entre le programme initial, purement fonctionnel, et le programme pivot fluxionnel afin de maintenir deux versions équivalentes du code source. En ajoutant à un programme écrit en Javascript son expression en Fluxions, l'équipe de développent peut le rendre *scalable* sans effort, tout en étant capable de répondre à la demande en fonctionnalités.

Ce travail s'est fait dans le cadre d'une thèse CIFRE dans la société Worldline. L'objectif pour Worldline est de se maintenir à la pointe dans le domaine du développement et de l'hébergement logiciel à travers une activité de recherche. L'objectif pour l'équipe Dice est de conduire une activité de recherche en partenariat avec un acteur industriel.

Contents

Chapter 1

Introduction

Contents

When the amazed 7 years old I was laid eyes on the first family computer, my life goal became to know everything there is to know about computers. This thesis is a mild achievement. It compiles my PhD work on *bridging the gap between development scalability and performance scalability, in the case of real-time web applications*.

illustration: amazed child in front of a computer

This work is the fruit of a collaboration between the Worldline company and the Inria DICE team (Data on the Internet at the Core of the Economy) from the CITI laboratory (Centre d'Innovation en Télécommunications et Intégration de services) at INSA de Lyon. For Worldline, this work falls within a larger work named Liquid IT, on the future of the cloud infrastructure and development. As defined by Worldline, Liquid IT aims at decreasing the time to market of a web service, allows the development team to focus on service specifications rather than technical optimizations and ease maintenance. The purpose of this PhD work, was to separate development scalability from performance scalability, to allow a continuous development from prototyping phase, until runtime on thousands of clusters. On the other hand, the DICE team focuses on the consequences of technology on economical and social changes at the digital age. This work falls within this scope as it studies the relation between the economical and the technological constraints driving the development of web services.

1.1 Web development

The growth of web platforms is partially due to Internet's capacity to allow very quick releases of a minimal viable product (MVP). In a matter of hours, it is possible to release a prototype and start gathering a user community around. *"Release early, release often"*, and *"Fail fast"* are the punchlines of the web entrepreneurial community. It is crucial for the prosperity of such project to quickly validate that the proposed solution meets the needs of its users. Indeed, the lack of market need is the first 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.

1.2 Performance requirements

If the service successfully complies with users requirements, its community might grow with its popularity. The service is scalable when it can quickly respond to this

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

growth. 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.

Once split, the service parts are connected by an asynchronous messaging system. Many tools have been developed to express and manage these parts and their communications. However, these tools impose specific interfaces and languages, different from the initial monolithic approach. It requires the development team either to be trained or to hire experts, and to start over the initial code base. This shift causes the development team to spend development resources in background without adding visible value for the users. It is a risk for the evolution of the project as the second and third reasons for startup failures are running out of cash, and missing the right competences.

1.3 Problematic and proposal

These shifts are a risk for the economical evolution of a web application by disrupting the continuity of its development process. The main question addressed by this thesis is how to avoid these shifts, so as to allow a continuous development? That is to reconcile the reactivity required in the early stage of development and the performance increasingly required with the growth of popularity. To answer this question, this thesis proposes a solution based on an equivalence between two different programming paradigms. On one hand, there is the imperative, functional, asynchronous programming model, embodied by Javascript. On the other hand, there is the dataflow, distributed, programming model, embodied by the concept of fluxions introduced in chapter 5.

This thesis contains two main contributions. The first contribution is a compiler allowing to split a program into a pipeline of stages depending on a common memory store. The second contribution, stemming from the first one, is a second compiler, allowing to make independent the stages of this pipeline. With these two contributions, it is possible to build a compiler that links an imperative representation with a flow-based representation. The imperative representation carries the functional modularization of the application, while the flow-based representation carries its execution distribution. A development team shall then use these two representations to continuously iterate over the implementation of an application, and reach both maintainability and performance.

1.4 Thesis organization

This thesis is organized in four main chapters. Chapter 2 introduces the context for this thesis and explains in greater details its objectives. It presents the challenge to build web applications at a world wide scale, without jamming the organic evolution of its implementation. It concludes drawing a first answer to this challenge. Chapter 3 presents the works surrounding this thesis, and how they relate to it. It defines into the notions outlined in the precedent chapter to help the reader understand better the context. The end of this chapter presents clearly the problematic addressed in this thesis. Chapter 4 presents the first contribution allowing to represent a program as a pipeline of stages. It introduces Dues to encapsulate these stages, based on Javascript Promises. Chapter 5 presents the second contribution allowing to make these stages independent. It introduces Fluxion to encapsulate these stages. Chapter 8 concludes this thesis, and draws the possible perspectives beyond this work.

Chapter 2

Context and objectives

Contents

This chapter presents the general context for this work, and leads to a definition of the scope of this thesis. Section 2.1 presents the context of web development, and the motivations that led the web to become a software platform. It presents Javascript as the trending language currently taking over web development. Then, it presents the challenges of developing web servers for large audiences. Section 2.2 states the problem tackled by this thesis, and its objectives. In the economical context of the Web, the languages often fail to grow with the project they initially supported very efficiently. The inadequacy of the languages to support the growth of web applications leads to wasted development efforts, and additional costs. The objective of this thesis is to avoid these efforts and costs. It intends to provide a continuous development from the initial prototype up to the releasing and maintenance of the complete product.

2.1 The Web as a Platform

Similarly to operating systems, Web browsers started as software products with extension capabilities with scripts and applications. The distribution of an applications is limited only by the platform it can be deployed on. The Web spreads the scalability of software distribution world wide with a near zero latency. It eventually became the main distribution medium, and the wider market there can possibly be for software. It led the Web to become a major platform, replacing operating systems.

Now, with web services, or Software as a Service (SaaS), the distribution medium is so transparent that owning a software product to have an easier access is no longer relevant. It stimulates a completely new business model based on an instantaneous and free access for the user, while claiming value for their data. The next paragraphs present Javascript, the language that allowed this new business model to emerge.

2.1.1 Javascript, The Language of the Web

In the 80's with Moore's law predicting exponential increase in hardware performance, reducing development time became more profitable than reducing hardware costs. Higher-level languages replaced lower-level languages, because the economical gain in development time compensated the decrease in performance. Most of the now popular programming languages were released at this time, Python(1991), Ruby(1993), Java(1994), PHP(1995) and Javascript(1995).

Java thrived in the software industry. However, it then lose the hype that drove the community innovation and creativity, and now struggles to keep up with the latest trends in software development. On the contrary, Ruby on Rails emerged from an industrial context, but is now open source, and backed by a strong community that makes it evolve and mature. Other languages like Python and PHP, emerged within a strong community, and were later adopted by the industry for web development. Django, the Python web frameworks, is used to develop many web applications in industrial contexts. And Wordpress, a PHP publishing platform, is an economical success. These examples show that the involvement of the community is critical for the adoption, evolution and maturation of a language.

Since a few years, Javascript is slowly becoming the main language for web development. It is the only choice in the browser. This position became an incentive to make it fast (V8, ASM.js) and convenient (ES6, ES7). And since 2009, it is present on the server as well with Node.js This omnipresence became an advantage. It allows to develop and maintain the whole application with the same language.

2.1.1.1 Historical Context

"There are only two kinds of languages: the ones people complain about and the ones nobody uses"

— B. Stroustrup[1]

Javascript was created by Brendan Eich at Netscape around May 1995, and released to the public in September. At the time, Java was quickly adopted as the default language for web servers development, and everybody was betting on pushing Java inside the browser as well. The history proved them wrong[2].

Javascript was released as a scripting engine in Netscape Navigator and later in its concurrent, Internet Explorer. The competition between the two was fragmenting the Web. So that Web pages could not be universal, and had to be designed for a specific browser. To stop this fragmentation, Netscape submitted Javascript to Ecma International for standardization in November 1996. ECMA International released ECMAScript – or ECMA-262 – in June 1997. The first standard for Javascript to which all browsers should refer for their implementations.

illustration: the ugly duckling

[1] `http://www.stroustrup.com/bs_faq.html#really-say-that`

[2] Except for mobiles, with the success of Dalvik Android. Though, firefox OS is a promising alternative using Javascript.

The initial release of Javascript was designed in a rush, within 10 days, and targeted unexperienced developers. For these reasons, the language was considered poorly designed and unattractive by the developer community.

Why does Javascript suck?[3] Is Javascript here to stay?[4] Why Javascript Is Doomed.[5] Why JavaScript Makes Bad Developers.[6] JavaScript: The World's Most Misunderstood Programming Language[7] Why Javascript Still Sucks[8] 10 things we hate about JavaScript[9] Why do so many people seem to hate Javascript?[10]

But this situation evolved drastically since. All web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime [26]. Any Javascript code is open, allowing the community to pick, improve and reproduce the best techniques [11]. Javascript is distributed freely, with all the tools needed to reproduce and experiment on the largest communication network in history. All these reasons made the popularity of the Web and Javascript.

"When JavaScript was first introduced, I dismissed it as being not worth my attention. Much later, I took another look at it and discovered that hidden in the browser was an excellent programming language."

— Douglas Crockford[12]

Javascript was initially limited to short interactions on web pages. The typical usage was to pre-validate forms on the client to avoid wasting bandwidth with wrongly formated requests to the server. This situation hugely improved since the beginning of the language. Nowadays, there is a lot of web-based applications replacing desktop applications, like mail client, word processor, music player, graphics editor...

ECMA International released several versions in the few years following the creation of Javascript. The third version contributed to give Javascript a more complete

[3] http://whydoesitsuck.com/why-does-javascript-suck/

[4] http://www.javaworld.com/article/2077224/learn-java/is-javascript-here-to-stay-.html

[5] http://simpleprogrammer.com/2013/05/06/why-javascript-is-doomed/

[6] https://thorprojects.com/blog/Lists/Posts/Post.aspx?ID=1646

[7] http://www.crockford.com/javascript/javascript.html

[8] http://www.boronine.com/2012/12/14/Why-JavaScript-Still-Sucks/

[9] http://www.infoworld.com/article/2606605/javascript/146732-10-things-we-hate-about-JavaScript.html

[10] https://www.quora.com/Why-do-so-many-people-seem-to-hate-JavaScript

[11] http://blog.codinghorror.com/the-power-of-view-source/

[12] http://javascript.crockford.com/survey.html

and solid base as a programming language. From this point on, the considerations for Javascript kept improving.

In 2005, James Jesse Garrett released *Ajax: A New Approach to Web Applications* [29]. Ajax uses Javascript to dynamically reload the content inside a web page, hence improving the user experience. It allows Javascript to develop richer applications inside the browser, from user interactions to network communications. The first web applications to use Ajax were Gmail, and Google maps[13]. The community released Javascript framework to assist the development of these larger applications. Prototype[14] and DOJO[15] are early famous examples, and later jQuery[16] and underscore[17].

illustration: Javascript with superpowers

In 2004, the Web Hypertext Application Technology Working Group[18] was formed to work on the fifth version of the HTML standard. The name is misleading, it is really about giving Javascript superpowers like geolocation, storage, audio, video, and many mores. The releases of HTML5 and ECMAScript 5, in 2008 and 2009, represent a milestone in the development of web-based applications. Around the same time, Google released the Javascript interpreter V8 for its browser Chrome, improving drastically the execution performance. Javascript became the *de facto* programming language to develop on this rising application platform that is the Web[19].

Javascript is widely used on the web, in open source projects, and in the software industry. With the increasing importance of client web applications, Javascript is assuredly one of most important language in the times to come. Moreover, Javascript allows to build the server side of web applications as well. The next paragraphs presents the event-loop model used to develop Javascript web applications, both client and server-side.

2.1.1.2 Event-Loop Execution Model

Javascript is often associated with an event-based paradigm to react to concurrent user interactions. In 2009, Joyent released Node.js to build real-time web services

[13] A more in-depth analysis of the history of Ajax, given by late Aaron Swartz `http://www.aaronsw.com/weblog/ajaxhistory`

[14] `http://prototypejs.org/`

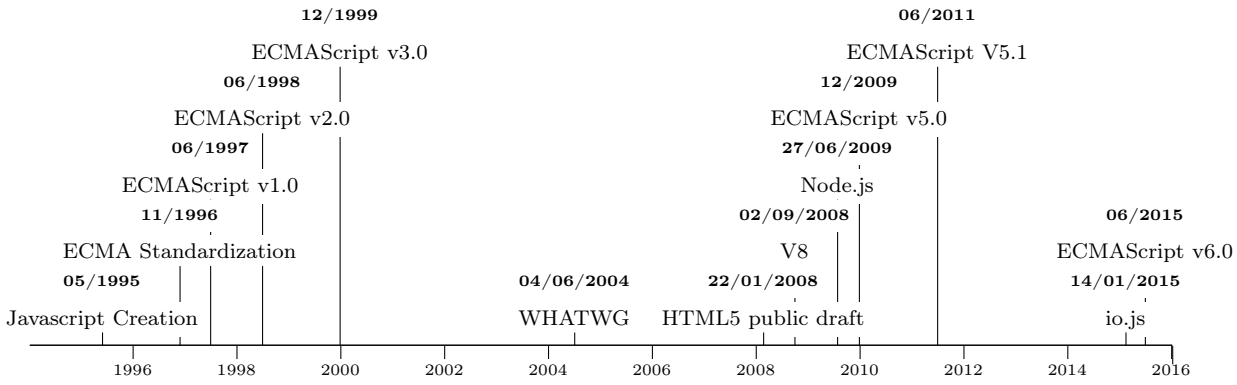
[15] `https://dojotoolkit.org/`

[16] `https://jquery.com/`

[17] `http://underscorejs.org/`

[18] `https://whatwg.org/`

[19] `http://blog.codinghorror.com/javascript-the-lingua-franca-of-the-web/`


12/1999
ECMAScript v3.0
06/1998
ECMAScript v2.0
06/1997
ECMAScript v1.0
11/1996
ECMA Standardization
05/1995
Javascript Creation
04/06/2004
WHATWG
06/2011
ECMAScript V5.1
12/2009
ECMAScript v5.0
27/06/2009
Node.js
02/09/2008
V8
22/01/2008
HTML5 public draft
06/2015
ECMAScript v6.0
14/01/2015
io.js
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016

with this paradigm. It is a server-side implementation of Javascript based on an event-loop. This event-based paradigm proved to be very efficient as well for a web service to react to concurrent requests. This section presents the event-loop execution model, and the advantages of Javascript for this paradigm.

The event-loop efficiency comes from non-blocking communications, asynchronous execution, and cooperative scheduling. It relies on a queue storing the messages received asynchronously. The loop executes tasks previously defined to process these messages one after the other. Each task can initiate new communications, leading in turn to the queuing of more messages, which trigger more tasks, and so on. Each task is executed atomically and exclusively, until it yields execution to continue with the next task in queue.

*

⚠ TODO schema of an event-loop

Callbacks In Javascript, the asynchronous communications are initiated by function calls. The callee immediately returns to avoid the caller to wait for the result. The task to process the result of the communication, and to continue the execution afterward, is a function passed as an argument to the callee. This function is named a callback or a continuation. A callback is a function passed as an argument to a callee. It is a continuation if the callee calls it to transfer the control back to the caller without the need for synchronization.

In this execution model, the control flow follows the asynchronous function calls and callbacks. It organizes the execution of callbacks causally, one after the other, similarly to a pipeline. Indeed, the input stream of data flows through a sequence of callbacks until the application outputs it. In this model, callbacks are the atoms of the asynchronous execution flow control. The next paragraph presents a more elaborate form of control.

Promises Since the asynchronous execution flow became more complicated on larger web application, many projects proposed improved asynchronous execution controls on top of callbacks. The ECMAScript specification proposes Promises for such purpose. It arranges sequences of causally related callbacks into cleanly organized pipelines of callbacks communicating their results to the next.

Closures Because callbacks can be passed as an argument, they are first class citizen and imply higher-order programming. For a callback to continue the execution without needing synchronization with the caller, it needs to have access to the initial context of the caller. This context is linked with the function when passed to the callee. The association of a function and its initial context is called a closure.

Higher-order programming is convenient for developers, as they allow great modularity in the implementation through *e.g.* inversion of control. It is presented in further details in section 3.1. However, because the contexts are passed, and shared all over the implementation, this programming model needs a global memory for coordination. A global memory is problematic to increase the concurrency of the execution.

This section presented Javascript as the language of the web, and its programming model. The next section presents the realities and technical challenges to assure the performance of web services against billions of users.

2.1.2 Highly Concurrent Web Servers

The previous section presented Javascript, the prolific language to build the Web. With SaaS, a Web service can scale world wide with near zero latency. With this broad range of distribution, a new business model emerged, allowing instantaneous and free access for the user. The usage exploded, and the software industry needed innovative solutions to cope with large network traffic.

illustration: busy web servers

2.1.2.1 Scalable Concurrency

The Internet allows communication at an unprecedented scale. There is more than 16 billions connected devices, and it is growing fast[20] [**Hilbert2011**]. A large web application like google search receives about 40 000 requests per seconds[21]. Such a Web application needs to be highly concurrent to manage this amount of simultaneous requests. In the 2000s, the limit to break was 10 thousands simultaneous connections with a single commodity machine[22]. In the 2010s, the limit is set at 10 millions simultaneous connections[23]. With the growing number of connected devices on the internet, concurrency is a very important property in the design of web applications. Moreover, the concurrency needs to be scalable to adapt to this growth of audience, as explained in the next paragraph.

Scalability The traffic of a popular web application such as Google search remains stable because of its popularity. The importance of the average traffic softens the occasional spikes. However, the traffic of a less popular web application is much

[20] `http://blogs.cisco.com/news/cisco-connections-counter`
[21] `http://www.internetlivestats.com/google-search-statistics/`
[22] `http://www.kegel.com/c10k.html`
[23] `http://c10m.robertgraham.com/p/manifesto.html`

more uncertain. For example, it might become viral when it is efficiently relayed in the media. The load of the web application increases with the growth of audience. The available resources needs to increase to meet this load. This growth can be steady enough to plan the increase of resources ahead of time, or it might be erratic and challenging. An application is scalable, if it is able to spread over resources proportionally as a reaction to the increasing growth of audience.

2.1.2.2 Time-slicing and Parallelism

Concurrency is achieved differently on hardware with a single or several processing units. On a single processing unit, the tasks are executed sequentially, interleaved in time. While on several processing units, the tasks are executed simultaneously, in parallel. Parallel executions uses more processing units to reduce computing time over sequential execution.

If the tasks are independent, they can be executed in parallel as well as sequentially. This parallelism is scalable, as the independent tasks can stretch the computation on the resources so as to meet the required performance. However, the tasks within an application need to coordinate together to modify the application state. This coordination limits the parallelism and imposes to execute some tasks sequentially. It limits the scalability. The type of possible concurrency, sequential or parallel, is defined by the interdependencies of the tasks.

As explained in the previous section, Javascript requires a global memory to coordinate the execution of the callbacks. The event-loop is constrained within time-slicing concurrency to assure this coordination. This thesis argues that there exists an equivalence between the event-loop model and the pipeline execution model. The next section presents this parallel execution model.

2.1.2.3 Pipeline Execution Model

The pipeline execution model is composed of isolated stages communicating by message passing to leverage the parallelism of a multi-core hardware architectures. It is well suited for streaming application, as the stream of data flows from stage to stage. Each stage has an independent memory to hold its own state. As the stages are independent, the state coordination between the stages are communicated along with the stream of data.

*

TODO schema of a pipeline

Each stage is organized in a similar fashion than the event-loop presented in section 2.1.1.2. It receives and queues messages from upstream stages, processes them one after the other, and outputs the result to downstream stages. The difference is

that in the pipeline architecture, each task is executed on an isolated stage, whereas in the event-loop execution model, all tasks share the same queue, loop and memory store.

* The next section details further the incompatibility in their model and the resulting economical consequences.

⚠ TODO schema to compare the event-loop and the pipeline, and an introducing sentence.

2.2 An Economical Problem

With the rise of SaaS on the Web, the software industry are in charge of both the development and the execution of the software. The previous section presented these two aspects individually. This section presents the challenges encountered by conducting the two at such a large scale. It then focuses on the subject and defines the objectives of this thesis.

2.2.1 Disrupted Development

The economical context on the Web allows a project to grow from a very early stage to a large business. The economical constraints to meet are very different in the beginning and in the maturation of such project. In the early steps the constraints hold on the development. The project needs crucially to reduce development costs, and to release a first product as soon as possible. On the contrary, in the maturation of the project, the constraints hold on the performance. The product needs to be highly concurrent to meet the load of usage. The team needs to adapt to meet the different constraints, which implies a disruption in the evolution of the project. This section further details the reasons and consequences of this disruption.

2.2.1.1 Power-Wall Disruption

illustration: heating chipset / parallel chipsets

Around 2004, the speed of sequential execution on a processing unit plateaued[24]. Manufacturers reached what they called the *Power-wall*. They started to arrange transistors into several processing units to keep increasing overall performance while avoiding overheating problems. Therefore, the performance of the sequential execution plateaued as well. Parallelism is the only option to achieve high concurrency on this parallel hardware. But the isolation required by parallelism is in contradiction with the best practices of software development to achieve maintainability. This *Power-wall* leads to a rupture between performance and maintainability.

[24] https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/

2.2.1.2 Unavoidable Modularity

The best practices in software development advocate to gather features logically into distinct modules. This modularity allows a developer to understand and contribute to an application one module at a time, instead of understanding the whole application. It allows to develop and maintain a large code-base by a multitude of developers bringing small, independent contributions.

This modularity avoids a different problem than the isolation required by parallelism. The former intends to structure code to improve maintainability, while the latter improve performance through parallel execution. These two organizations are conflicting in the design of the application. The next paragraph presents the disruptions in the development of a web application implied by this conflict.

2.2.1.3 Technological Shift

Between the prototyping, and the maturation of a web application, the needs are radically different. During the initiation of a web application project, the economical constraint holds on the pace of development. The development reactivity is crucial to meet the market needs[25]. The development team opt for a popular and accessible language to leverage the advantage of its community. It is only after a certain threshold of popularity that the economical constraint on performance requirements exceeds the one on development. The development team then shifts to an organization providing parallelism.

This shift brings two risks. The development team needs to rewrite the code base to adapt it to a completely different paradigm. The application risks to fail because of this challenge. And after this shift the development pace slows down. The development team cannot react as quickly to user feedbacks to adapt the application to the market needs. The application risks to fall in obsolescence.

The risks implied by this rupture proves that there is economically a need for a solution that continuously follows the evolution of a web application. The next section presents the proposition of this thesis for such a solution. It would allow developers to iterate continuously on the implementation focusing simultaneously on performance, and on maintainability.

2.2.2 Seamless Web Development

This thesis is conducted in the frame of a larger work on LiquidIT within the Worldline company. Worldline develops and hosts real-time streaming Web services, and

[25] `https://www.cbinsights.com/blog/startup-failure-post-mortem/`

identified that one of their need was to increase the time to market for its products. Worldline defines LiquidIT as *a concept of flexible and cost-effective IT services that can be provisioned, built and configured in real time, allowing end-to-end financial transparency*. It precisely intends to provide *business agility, investment-free charging models, flexibility and ease of use*. The goal of this thesis in this larger work, is to allow the developer to focus solely on business logic, and leave the technical constraints of performance scalability to automated tools. This section presents the objective of this work to avoid the disruption in development, and provide a seamless development experience.

2.2.2.1 Real-Time Streaming Web Services

This thesis focuses on web applications processing streams of requests from users in soft real-time. Such applications receive requests from clients through the HTTP protocol and must respond within a finite window of time. They are generally organized as sequences of tasks to modify the input stream of requests to produce the output stream of responses. The stream of requests flows through the tasks, and is not stored. On the other hand, the state of the application remains in memory to impact the future behaviors of the application. This state might be shared by several tasks within the application, and imply coordination between them.

The next section introduces the similarities and differences between the two programming models from the previous section. And then draws an equivalence. This equivalence is developed throughout this thesis.

2.2.2.2 Differences

Both paradigms encapsulate the execution in tasks assured to have an exclusive access to the memory. However, they provide two different models to provide this exclusivity resulting in two distinct programming models. Contrary to the pipeline architecture, the event-loop provides a common memory store allowing the best practice of software development to improve maintainability.

However, these two organizations are incompatible. Because of economical constraints, this incompatibility implies ruptures in the development. It represents additional development efforts and important costs. This thesis argues that it is possible to allow a continuous development between the two organizations, so as to lift these efforts and costs. The argumentation of this possibility is based on an equivalence bridging the two organizations. This equivalence is presented briefly in the next paragraph, and detailed further in the the chapter 4 and 5.

2.2.2.3 Equivalence

In the beginning of a project, the team adopt the event-loop execution model to focus on maintainability and evolution, discarding the scalable performance concerns. And as the project gather audience and the performance concerns become more and more critical, the development team adopt the pipeline execution model to take into account this performance concerns. The equivalence would allow a compiler to transform an application expressed in one model into the other.

With this equivalence, it would be possible to express an application following the design principles of software development. A development team could rely on the common memory store of the event-loop execution model, and focuses on the maintainability of the implementation. And yet, because of the equivalence between these two models, the execution engine could adapt itself to any parallelism of the computing machine, from a single core, to a distributed cluster. The development team could continuously progress with the two models and take advantage of their different concerns about the implementation, performance and maintainability.

This thesis proposes to provide an equivalence between the two memory models for streaming web applications. The goal of conciliating these two concerns is not new. The next chapter presents all the previous results needed to understand this work, up to the latest advances in the field.

Chapter 3

Software Design, State Of The Art

Contents

"A designer is responsible for producing the greatest benefit for any given investment of time, talent, money, and other resources."
— K. Sullivan, W. Griswold, Y. Cai, B. Hallen [**Sullivan2001a**]

The growth of the web and Software as a Service (SaaS) revealed the importance of previously unknown economic constraints. The same company carries both development and exploitation of a service in scale of unprecedented size. Development costs are reduced by following best practice, and by building maintainable softwares. However, as seen in the previous chapter, it is compensated by increasing hardware performance, which eventually rises exploitation costs. Similarly, exploitation costs are reduced by following more efficient programming models. But again, it rises development costs. So a SaaS company needs to cleverly allocate its budget between development and exploitation so as to limit the overall cost.

Eventually, a company faces the problem of scalability limitations. Compensating development with hardware becomes unsustainable. The company has no choice but to commit huge development efforts to get correct performances. This chapter draws a broad view of the relation between the orientation of development toward maintainability or scalable performance, and its consequences.

The best practices in software design advocate to decompose a problem into many subproblems. The decomposition of the implementation of a problem improves directly its maintainability, development scalability and evolution. This chapter present some of these best practices, *e.g.* modular programming, structured design [74], hierarchical structure [23] and object-oriented programming.

A few decades ago, the best practices were not concerned with execution performance. Moore's law [58] was wrongly interpreted as the assurance that hardware could always increase execution speed. But eventually, the clock speed of processors plateaued [1][**Bohr2007**], and the processing units were organized as several execution units to continue improving performances. But this hardware improvement could not anymore increase the execution speed without any additional development effort.

The best practices of software design then inherited two goals : to assure a scalable implementation evolution by decomposing it into subproblems, as well as assure a scalable parallel execution by decomposing the execution onto the several execution units. As D. L. Parnas showed in 1972 [64], it seems challenging to develop a software following a decomposition that satisfies both goals.

The evolution of economic constraints often force an SaaS company to switch from a maintainable to a scalable implementation. This switch implies huge development efforts. There has been many attempts at reconciling the two goals to reduce these costs. But none seems really convincing enough to be widely adopted.

[1] `https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/`

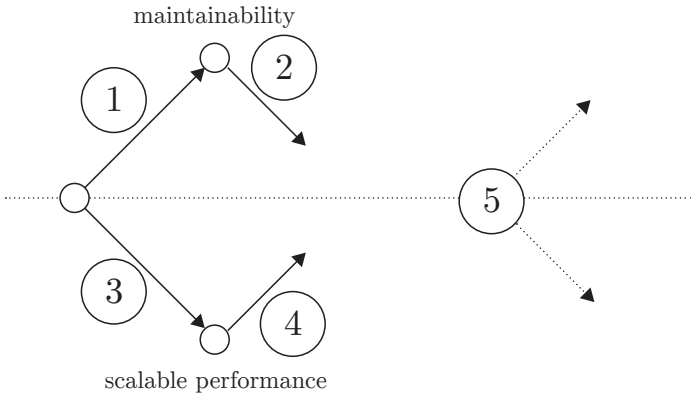


Figure **??** is a graphical representation of the organization of this chapter, and more generally of the state of the art of software design. The solutions focusing on maintainability, noted by number 1, are addressed in section 3.1 while the solutions focusing on performance scalability, noted by number 3, are addressed in section 3.2. Each of these direction of development contains works trying to meet the requirements from the opposing category, noted by number 2 and 4. The focus on both development and performance scalability is addressed in section **??**, noted by number 5. And finally, section **??** presents a synthesis of the state of the art presented in this chapter.

All the solutions presented throughout this chapter are analyzed against three criteria.

- Maintainability
- Organic Growth
- Performance Scalability

Maintainability rely on modular programming, which requires three criteria. They are explained in section 3.1.1.

- encapsulation mechanism
- presence of higher-order programming
- presence of lazy evaluation, or stream composition

For a solution to be maintainable in regard to the economic context previously stated, it needs an organic growth. It needs to be backed by a strong community, and industrial needs. And to be relevant in this context, it needs to support web technologies. These criteria are explained in section 3.1.2.

- adoption by the community
- adoption by the industry
- supporting web technologies

Performance scalability rely on parallelism. More precisely, the requirements for scalable performance are an blend of shared states at a fine level, and isolation at a coarse level. These criteria are explained in section 3.1.3.

- fine level sequentiality
- coarse level message passing

3.1 Software Maintainability

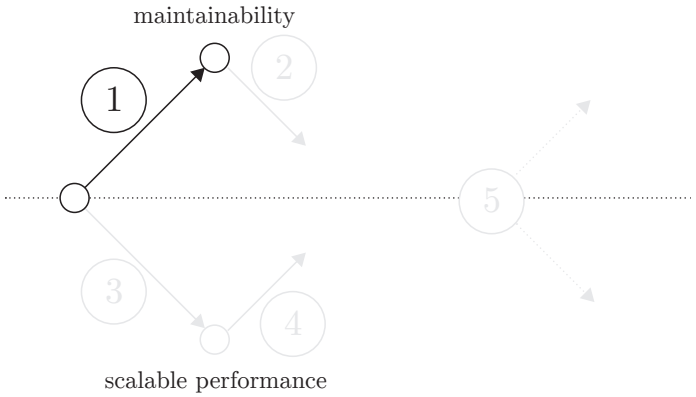
"It is becoming increasingly important to the data-processing industry to be able to produce more programming systems and produce them with fewer errors, at a faster rate, and in a way that modifications can be accomplished easily and quickly."
— W. P. Stevens, G. J. Myers, L. L. Constantine [74].

In order to improve and maintain a software system, it is important to holds in mind a mental representation of its implementation. As the system grows in size, the mental representation become more and more difficult to grasp. Therefore, it is crucial to decompose the system into smaller subsystem easier to grasp individually.

Section 3.1.1 presents the modular programming paradigms, and their programming models, oriented toward maintainability. Section 3.1.2 presents the organic growth allowed by the maintainability of modular programming. Section 3.1.3 presents the consequences of the modularity on performance. Finally, section 3.1.4 summarizes the three previous sections in a table.

3.1.1 Modularity

The modularity of a software implementation is about enclosing subproblems and composing them through relevant interface abstractions. It improves the maintainability of an implementation, as illustrated in figure **??**. It allows to limit the understanding required to contribute to a module [74]. And it reduces development



time by allowing several developers to simultaneously implement different modules [**Cataldo2006**, 80].

illustration: spaghetti programming

Modular Programming stands upon Structured Programming [**Dijkstra1970**]. It draws clear interfaces around a piece of implementation so that the execution is enclosed inside. At a fine level, it helps avoid spaghetti code [22], and at a coarser level, it structures the implementation [23] into modules, or layers. Encapsulate a specific design choice in each module, so that it is responsible for one and only one concern, isolate its evolution from impacting the rest of the implementation [**Tarr1999**, **Hursch1995**, 64]. Examples of such separation of concerns are the separation of the form and the content in HTML / CSS, or the OSI model for the network stack.

illustration: lasagna programming

The criteria to define modules to improve maintainability are coupling and cohesion [74]. The coupling defines the strength of the interdependence between modules, while cohesion defines how strongly the features inside a module are related. Low coupling between modules and high cohesion inside modules helps logically organize, and understand the implementation. Hence, it improve its maintainability.

The composition of modules with low coupling is provided by encapsulation, higher-order programming and lazy evaluation. Hence, the criteria to analyze the solutions presented in this section regarding maintainability are :

- encapsulation mechanism
- presence of higher-order programming
- presence of lazy evaluation, or stream composition

The last two criteria provide composition abstractions to design solutions from smaller components. Encapsulation and composition are the features in modular programming that produce maintainability.

The next paragraphs present the last two criteria 3.1.1.1, and then the main programming models, section 3.1.1.2.

3.1.1.1 Features

Higher-Order Programming *

Higher-order programming allows to manipulate functions like any other primary value : to store them in variables, or to pass them as arguments. It replaces the need for most modern object oriented programming design patterns [2] with Inversion of Control [**Johnson**], the Hollywood Principle [**Sweet1985**], and Monads [**Wadler1992**]. Higher-order programming help loosen coupling, thus improve maintainability.

In languages allowing mutable state, higher-order functions are implemented as closure, to preserve the lexical scope [75]. A closure is the association of a function and a reference to the lexical context from its creation. It allows this function to access variable from this context, even when invoked outside the scope of this context. * It eventually tangles the memory references so that it requires a global memory.

Lazy Evaluation Lazy evaluation is an evaluation strategy allowing to defer the execution of a function only when its result is needed. The lazy evaluation of lists is equivalent to a stream with a null-sized buffer, while the opposite, eager evaluation, corresponds to an infinite buffer [**VanRoy2003**]. *Indeed, the dataflow programming paradigm resulting from lazy lists is particularly adapted for stream processing applications.

* The lazy evaluation, as well as streams are powerful tools for structuring modular programs [**Sussman1983**]. Lazy evaluation allows the execution to be organized as a concurrent pipeline, as the stages are executed independently for each element of the stream. But this concurrency requires immutability of state, or at least isolation of side-effects.* The next section addresses the consequences of higher-order programming and lazy evaluation on parallelism.

[2] http://stackoverflow.com/a/5797892/933670

If possible, include this reference : Continuations and coroutines [36]

next sentence is redundant with the suit

find another transition

This paragraph is not very clear

why ? explain or point to the explanation

3.1.1.2 Programming Models

The next paragraphs presents the different programming model regarding their support to modular programming and maintainability.

Imperative Programming Imperative programming is the very first programming paradigm, as it evolves directly from the hardware architectures. It allows to express the suite of operation to carry sequentially on the computing processor. Most imperative languages provide encapsulation with modules but not higher-order programming, nor lazy evaluation.

illustration: multiple cells communicating

Object Oriented Programming The very first OOP language was Smalltalk [**Goldberg1984**]. It defined the core concepts of OOP as message passing, encapsulation, and late-binding[3]. Nowadays, the major emblematic figures of OOP in the software industry are C++ and Java [**Gosling2000**, **Stroustrup1986**]. They provide encapsulation with Classes, and higher-order programming with objects, but not lazy evaluation.

Functional Programming The definition of pure Functional Programming resides in manipulating only mathematical expressions - functions - and forbidding state mutability, replaced by message passing. The absence of state mutability makes a function side-effect free, hence their execution can be scheduled in parallel. But it implies heavy message passing, which negatively impact performances. The most important pure Functional Programming languages are Scheme [**Rees1986**], Miranda [**Turner1986**], Haskell [**Hudak1992**] and Standard ML [**Milner1997**]. They provide encapsulation, higher-order programming and lazy evaluation.

Multi-Paradigm The functional programming concepts are also implemented in other languages along with mutable states and object-oriented concepts. Major recent programming languages now commonly present **higher-order functions** and **lazy evaluation** to help loosen the couple between modules, define more generic and reusable modules. Moreover, they provide reflective programming, and other meta-programming techniques. *In fine*, it helps developers to write applications that are more maintainable, and favorable to evolution [**Hughes1989**, **Turner1981**]. These recent multi-paradigms languages like Javascript, Ruby, Python and Go combine the different paradigms to help developer building applications faster. They provide encapsulation, higher-order programming and stream composition.

[3] `http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en`

Table **??** presents a summary of the analysis of the programming models presented in the previous paragraphs.

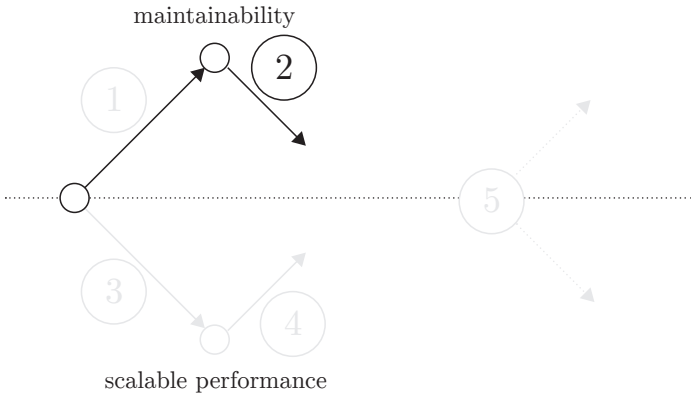
Model	Implementations	Higher-Order Programming	Lazy evaluation, or streaming composition	Encapsulation	→	Maintainability
Imperative Programming	C	✗	✗	✓		✗
Object-Oriented Programming	C++, Java	✗	✗	✓		✗
Functional Programming	Scheme, Miranda, Haskell	✓	✓	✓		✓
Dynamic Scripting	Javacript, Python, Ruby, Go	✓	✓	✓		✓

Table 3.1 – Analysis of the state of the art in modular programming regarding maintainability

3.1.2 Organic Growth

A system is maintainable only if there is people willing to maintain it. For people to widely adopt a programming language, it needs to be features a balance between performance and maintainability. It forces the modular programming models to be implemented taking into account not only maintainability, but performance as well. The balance is steered back toward performance, as illustrated in figure **??**.

Indeed, the immutability of pure functional programming impact performance too negatively to be wildly used in industrial context. And the pure object-oriented programming lacks composition abstractions, like higher-order programming and lazy



evaluation. As a result, the multi-paradigm scripts tends to be better compromises between modularity and performance. And the data of this section proves it.

The criteria to analyze the solutions presented in this section regarding the organic growth are the adoption in :

- the community,
- the industry,
- web technologies

The first two criteria make sure that the technology is growing organically with a passionate community, and backed by industrial needs. The last criteria assures the fitting of the technologies with our economical context of a web application. The next paragraphs presents different sets of data to emphasize the adoption of the previously presented programming model by the community and the industry.

3.1.2.1 Community

Available Resources According to the TIOBE Programming Community index, Javascript ranks 8th, as of October 2015, and was the most rising language in 2014. This index measure the popularity of a programming language with the number of results on many search engines. However, this measure is controversial as the number of pages doesn't represent the number of readers. Alternatively, Javascript ranks 7th on the PYPL, as of October 2015. The PYPL index is based on Google trends to measure the number of requests on a programming language. However, it is limited

to Google searches. From these indexes, the major programming languages are Java, then C/C++, C# and Python. The preponderance of Object-Oriented Languages in these results is explained by the lag of these indexes. Java and C/C++ were very popular and helped bring the explosion of the web. It is not unusual to see these languages ranked high as they are still widely used as the communities and the industry continue to use them. Though, the next paragraphs bring a more actual vision.

*

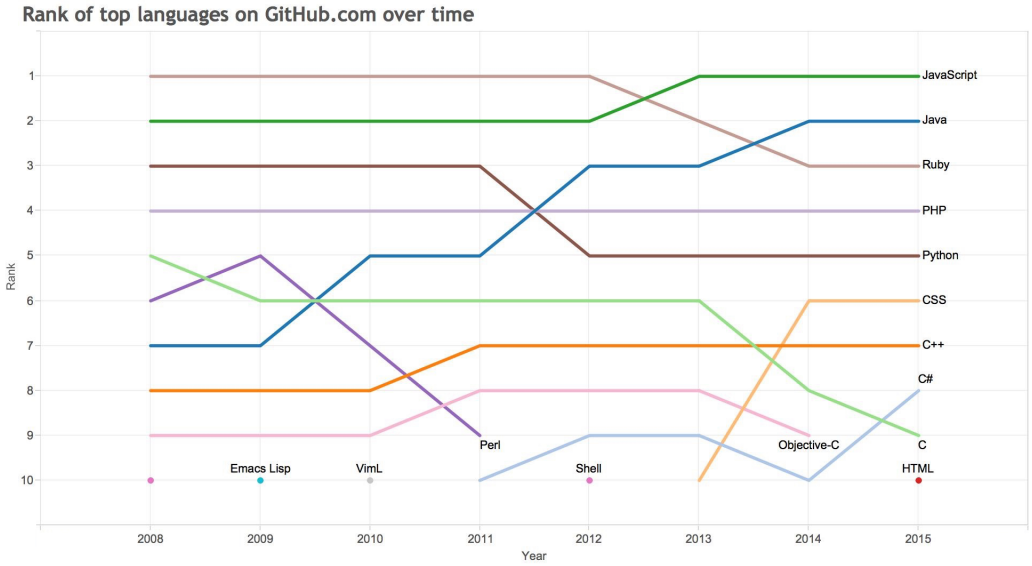
⚠ TODO graphical ranking of TIOBE and PYPL

Developers Collaboration Platforms Online collaboration tools give an indicator of the number of developers and projects using certain languages. Javascript is the most used language on *Github*, the most important collaborative development platform gathering about 9 millions users. It represents more than 320 000 repositories, while the second language is Java with more than 220 000 repositories. Javascript is the most cited language on *StackOverflow*, the most important Q&A platform for developers. It represents more than 960 000 questions, while the second is Java with around 940 000 questions. According to *Black Duck Software*[4] it is the second language used in open source projects. And according to a survey by *StackOverflow*, it is currently the language the most popular[5]. Moreover, the Javascript package manager, *npm*, has the most important and impressive package repository growth. *

⚠ TODO include so survey graph

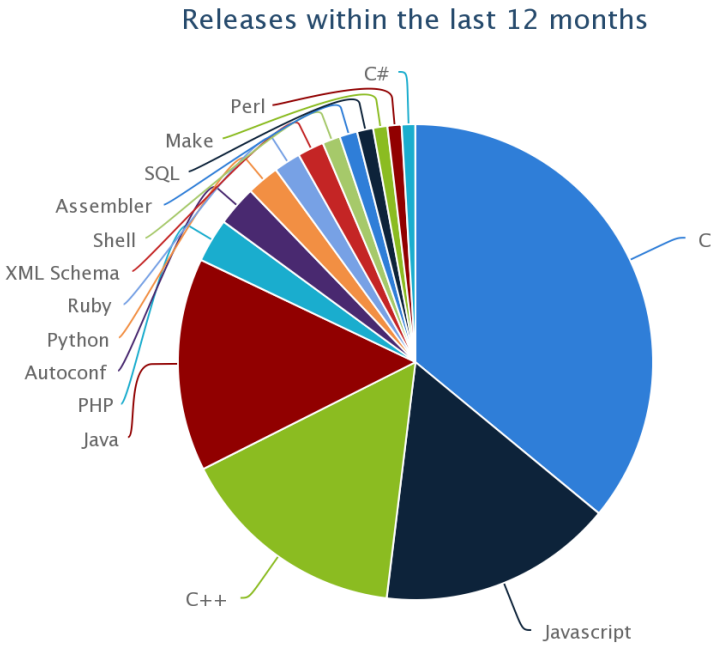
[4] https://www.blackducksoftware.com/

[5] http://stackoverflow.com/research/developer-survey-2015



[6]

TODO redo this graph, it is ugly.

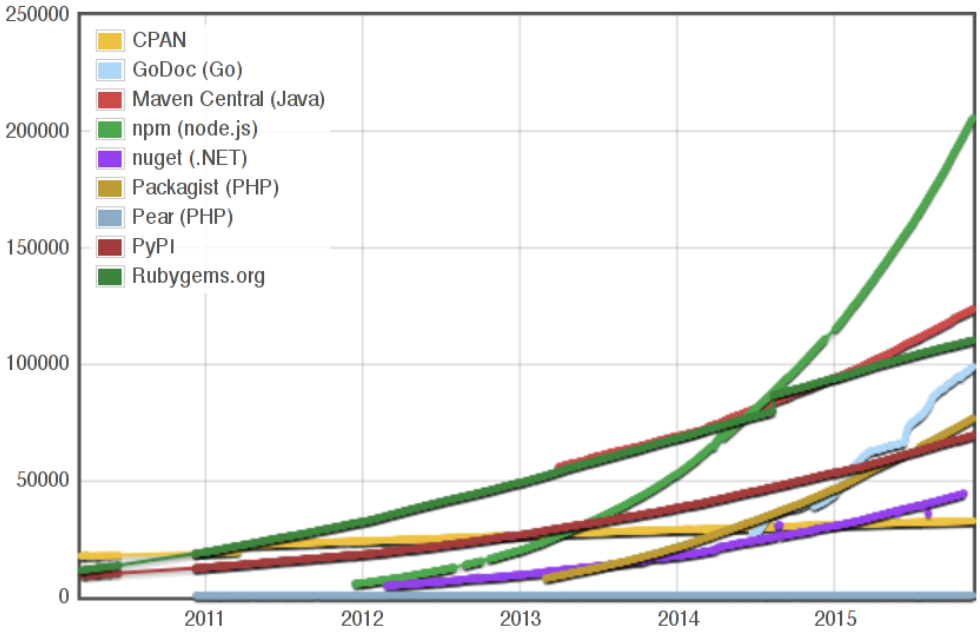


TODO graphical ranking of the tags in StackOverflow

Black Duck

TODO redo this graph, it is ugly.

[6] https://github.com/blog/2047-language-trends-on-github



*

⚠ TODO redo this graph, it is ugly.

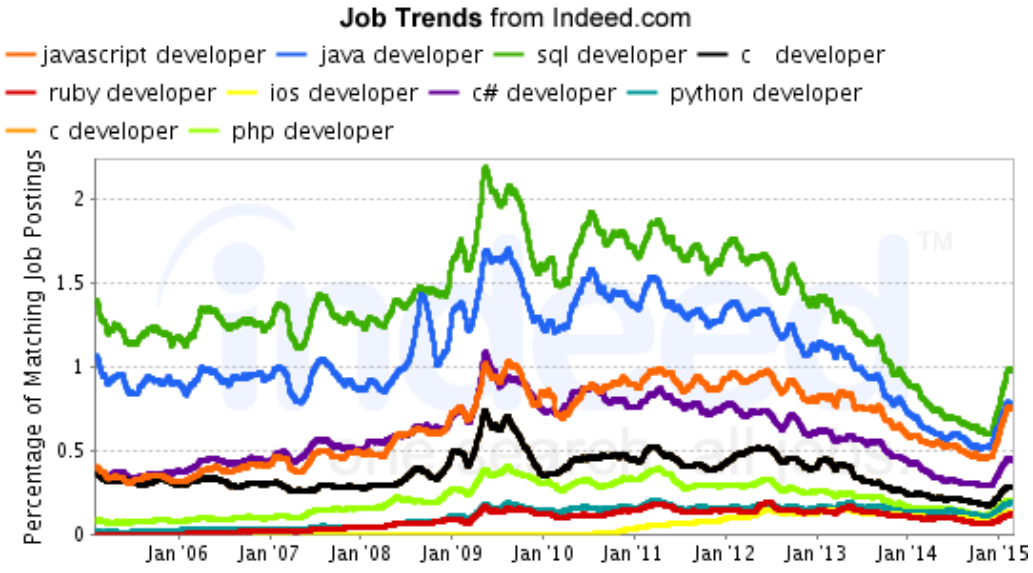
3.1.2.2 Industry

The actors of the software industry tends to hide their activities trying to keep an edge on the competition. The previous metrics represent the visible activity but are barely representative of the software industry. The trends on job opportunities give some additional hints on the situation. Javascript is the third most wanted skill, according to *Indeed*[7], right after SQL and Java.[8] Moreover, according to *breaz.io*[9], Javascript developers get more opportunities than any other developers. Javascript is increasingly adopted in the software industry.

[7] `http://www.indeed.com`

[8] `http://www.indeed.com/jobtrends?q=Javascript%2C+SQL%2C+Java%2C+C%2B%2B%2C+C%2FC%2B%2B%2C+C%23%2C+Python%2C+PHP%2C+Ruby&l=`

[9] `https://breaz.io/`



TODO redo this graph, it is ugly.

All these metrics represent different faces of the current situation of the Javascript adoption in the development community and industry. It is widely used on the web, in open source projects, and in the software industry. With the increasing importance of client web applications, Javascript is assuredly one of most important language in the times to come. Moreover, Javascript allows to build the server side of web applications as well. The next paragraphs presents the event-loop model used to develop Javascript web applications, both client and server-side.

3.1.3 Performance Limitations

Eventually, the presented languages are hitting a wall on their way to performance. The next paragraph briefly present the encountered incompatibility between modularity and performance to introduce the next section of this chapter.

Parallelizing the execution increases the performances [7, 32] But it is limited because accesses to sharing state need to be scheduled sequentially. Hence, to increase the parallelism and performance, the concurrent executions need to be independent, or to coordinate to be scheduled sequentially [34, 33, 61, 31]. On the other hand, completely forbidding shared state, with immutable state, increase the need for communication because of the additional required synchronization. Eventually, the communications induce a greater overhead than the performance increases from parallelization, and it finally negatively impacts the performances.

To be scalable, a solution needs to allow shared state at a fine grain, where

Model	Implementations	Community adoption	Industrial adoption	Web compliant	→	Growth
Imperative Programming	C	✓	✓	✗		✗
Object-Oriented Programming	C++, Java	✓	✓	✓		✓
Functional Programming	Scheme, Miranda, Haskell	✗	✗	✗		✗
Dynamic Scripting	Javacript, Python, Ruby, Go	✓	✓	✓		✓

Table 3.2 – Analysis of the state of the art in modular programming regarding organic growth

lots of synchronization is required, and where communication would induce a greater overhead than parallelization could compensate. And immutability at a coarse grain, where lesser synchronization is required, and parallelization can be advantageously used.

Encapsulation aims not to provide this decomposition between sharing and immutable space required for performance scalability. It aims to draw a clear boundary around the concern of a module to help understanding it. To allow higher-order programming and mutable state, despite encapsulation, languages implement closures, and intermingle the memory between modules. It reinforces the need for synchronization and sequentiality.

Finally, the requirement to enforce the decomposition required for performance scalability are :

- shared state : fine level sequentiality
- immutable state : coarse level message passing

3.1.4 Summary

Table **??** summarizes the characteristics of the solutions presented in this section.

Model	Implementations	Shared state	Immutable state	→	Scalability
Imperative Programming	C	✓	✗		✗
Object-Oriented Programming	C++, Java	✓	✗		✗
Functional Programming	Scheme, Miranda, Haskell	✗	✓		✗
Dynamic Scripting	Javacript, Python, Ruby, Go	✓	✗		✗

Table 3.3 – Analysis of the state of the art in modular programming regarding scalability

Model	Implementations	Maintainability	Organic Growth	Scalability
Imperative Programming	C	✗	✗	✗
Object-Oriented Programming	C++, Java	✗	✓	✗
Functional Programming	Scheme, Miranda, Haskell	✓	✗	✗
Dynamic Scripting	Javacript, Python, Ruby, Go	✓	✓	✗

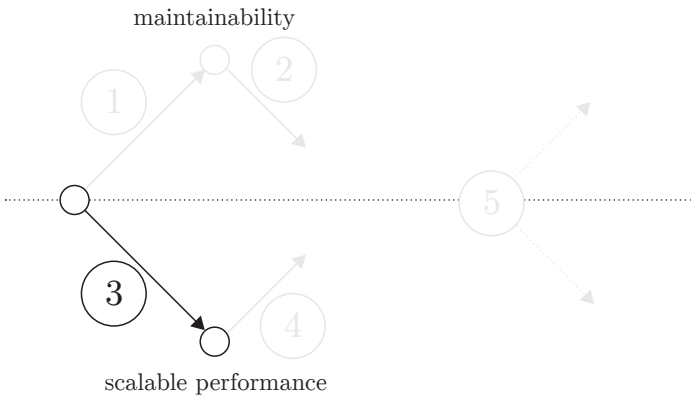
Table 3.4 – Synthesis of the state of the art in modular programming

3.2 Software Performance

With the limitations on performance presented in the previous section, both academy and industry communities proposed alternative solutions with performance in mind. Section 3.2.1 presents the concurrent and parallel programming paradigms, and their programming models oriented on performance rather than maintainability. Section 3.2.2 presents the organic growth steered by the performance of parallel program-

ming. Section 3.2.3 presents the consequences of the decomposition on maintainability. Finally, section 3.2.4 summarizes the three previous sections in a table.

3.2.1 Concurrency



Web servers need to be able to process huge amount of concurrent tasks in a scalable fashion. Concurrency is the ability to make progress on several tasks simultaneously, it implies decomposition of the execution or the memory into independent tasks. When the independence on both the execution and the memory are clearly defined, the tasks can be scheduled in parallel, otherwise, they cannot.

This decomposition allows the fine level sequentiality, and coarse level message passing required by scalability. The sequentiality of execution at a fine level assures the invariance on the shared state, and avoid communication overhead. The message passed at a coarser level are immutable, it assures the independence, hence the parallelism, and avoid the synchronization overhead. The two are indispensable for performance scalability as the need for synchronization are different at the two levels. Therefore the solutions presented in this section are analyzed against these two criteria.

- fine level sequentiality
- coarse level message passing

3.2.1.1 Concurrent Programming

illustration: feu rouge et rond point

Concurrent programming provides to the developer the mechanisms to decompose the execution for concurrency, while conserving a global memory model.

There are two scheduling strategies to execute tasks sequentially on a single processing unit, cooperative scheduling and preemptive scheduling. Cooperative scheduling allows a concurrent execution to run until it yields back to the scheduler. Each concurrent execution has an atomic, exclusive access on the memory. On the other hand, preemptive scheduling allows a limited time of execution for each concurrent execution, before preempting it. It assures fairness between the tasks, such as in a multi-tasking operating system, but as the preemption happens unexpectedly it breaks atomicity, the developer needs to lock the shared state to assure atomicity and exclusivity. The next paragraphs presents the event-driven programing model, based on cooperative scheduling, and the multi-threading programming model, based on preemptive scheduling. Additionally, they present lock-free data-structures, which is independent from the scheduling strategy, as they rely on atomic memory operations.

Event-Driven Programming Event-driven programming explicitly queues the concurrent executions needing access to shared resources. The concurrent executions are schedule sequentially to assure exclusivity, and cooperatively to assure atomicity.

As presented in the previous chapter, web servers needs to be highly concurrent, and efficient. The event-driven model is very efficient to serve websites, as it avoids contention due to waiting for shared resources like disks, or network. Web servers like Flash [63], Ninja [30] thttpd[10] and Nginx[11] were designed following this model. However, a drawback of this model was that the execution context is lost at each event. The developer needs to explicitly transfer the relevant state to continue the execution from one event execution to another.

Cooperative threads, or fibers, addressed this drawback [4, 12]. The execution is not ripped into several events. It yields and resume exactly at the same point after the completion of an asynchronous operation, conserving its context. However, the developer needs to be well aware of the asynchronous calls to assure the atomicity[12].*

⚠ more about that, it is important, and transition with the next

The problem of losing the execution context disappears with closures in higher-order programming. * Moreover, the continuation passing style used in higher-order programming requires the developer to be aware of the asynchronous rupture in

⚠ link with the previous paragraph

[10] http://acme.com/software/thttpd/

[11] https://www.nginx.com/

[12] https://glyph.twistedmatrix.com/2014/02/unyielding.html

the execution, so as to assure atomicity [75]. And because an asynchronous call doesn't wait for the completion of the operation, the asynchronous control flow is not limited to be linear like in threads. * Multiple asynchronous calls are made in parallel. Several execution model proposed this event-based programming model, like TAME [49], Node.js[13] and Vert.X[14].

However, as the shared memory is global and all the execution portions needs atomic access, they are not parallel, but sequentially concurrent. The next paragraph present work intending to improve performance by reducing the sequential portions to a minimum to increase the possibilities of parallelism.

Lock-Free Data-Structures The wait-free and lock-free data-structures reduce the exclusive execution to a few atomic operations [**Lamport1977**, **Herlihy1988**, **Herlihy1990**, **Herlihy1991**, **Anderson1990**]. They are based on transactional memories [**Harris2010**], which provide atomic read and write operations on a shared memory. Lock-free data-structures arrange these atomic operations so as to keep invariance without the need to lock. They provide concurrent implementation of basic data-structures such as linked list [**Valois1995**, **Timnat2012**], queue [**Sundell2003**, **Wimmer2015**], tree [**Ramachandran2015**] or stack [**Hendler2004**].

However, even if they are theoretically infinitely scalable, they are hard to come with, and are not fit for every problem. The next paragraph present the multi-threading and associated synchronization mechanisms to try improve the parallelism of execution using coarser granularity of atomic execution and exclusivity.

Multi-Threading Programming Threads are light processes sharing the same memory execution context within an isolated process [23]. They wait for completion of each operation, and are preemptively scheduled to avoid blocking the global progression. This preemption breaks the atomicity of the execution, and the parallel execution breaks the exclusivity. To restore atomicity and exclusivity, hence assure the invariance, multi-threading programming model provide synchronization mechanisms, such as semaphores [**Dijkstra**], guarded commands [21], guarded region [**Hansen1978a**] or monitors [41]. They assure an execution region to have exclusive access over a cell of the global state.

The purpose of explicit synchronization is to manage the timing of side-effects in the presence of parallelism.[15]

[13] `https://nodejs.org/en/`

[14] `http://vertx.io/`

[15] `http://pchiusano.blogspot.com/2010/01/actors-are-not-good-concurrency-model.html?showComment=1267337235223#c3014836700278061280`

*

Developers tend to use the global memory extensively, and threads require to protect each and every shared memory cell. This heavy need for synchronization leads to bad performances, and is difficult to develop with [4].

⚠ Fork-join parallelism : Cilk [**Randall1998**, **Frigo1998**, **Leiserson2010**]

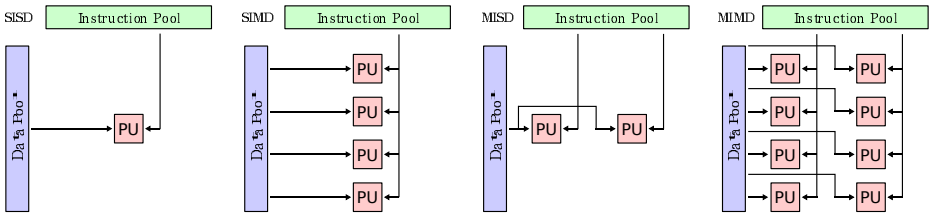
Model	Implementations	Fine level sequentiality	Coarse level message passing	→	Scalability
Event-driven programming	Node.js, Vert.X	✓	✗		✗
Lock-free Data Structures		✓	✗		✗
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✓	✗		✗

Table 3.5 – Analysis of the state of the art in concurrent programming regarding performance

Moore's law [58] which forecasts the density of transistors per processing unit, was wrongly interpreted to promise the exponential evolution in the sequential performance of the processing unit, and the assurance for the software industry of always faster hardware. But as transistors attained a critical size, the reduction in power required by transistor predicted by the Dennard's MOSFET scaling [**Dennard2007**] stopped[16]. The ever growing number of transistor predicted by Moore's law are arranged in parallel architecture to continue increasing the performance of processing units. Parallel programming became the only solution for scalable performance, at the expense of development effort.

Concurrent programming is a compromise to keep shared states, while introducing different forms of synchronization to keep the execution sequential. But because of

[16] `https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/`



by I, Cburnett. Licensed under CC BY-SA 3.0 via Commons
`https://commons.wikimedia.org/wiki/File:{SISD,SIMD,MISD,MIMD}.svg`

Figure 3.1 – Flynn's taxonomy of parallelism

the lack of clear isolation, the execution is not parallel, hence the performance is limited.

3.2.1.2 Parallel Programming

Concurrent programming is based on the causal ordering of execution. The ordering of operations is local within a synchronous execution, while the concurrent executions are causally ordered. It leads to parallel execution with some coordinations such as synchronization, immutability or isolation. As Lamport showed [50], and Reed related later [68], This causal order is sufficient to execute correctly a system in parallel, such as in distributed system.

This section presents first hand the theoretical and programming model based on asynchronous communication and isolated execution for parallel programming, as illustrated on the schema above. It then continues with stream processing programming model. And finally,it concludes on the limitations of parallel programming regarding accessibility.

*

⚠ read and include [c2006]

Asynchronous and Isolated Process Parallelism The Flynn's taxonomy **[Flynn1972]** is the most commonly used to categorize parallel execution. It separates the flow of instructions, and the flow of data ; each being unique, or multiple. All the current parallel programming model currently belong to the category Multiple Instruction Multiple Data (MIMD), which is further divided into Single Program Multiple Data (SPMD) [**Auguin1983**, **Darema1988**, **Darema2001**] and Multiple Program Multiple Data (MPMD) [**Chang1997**, **Chan2004**]. MIMD implies several threads of execution processing several stream of data.

The difference between SPMD and MPMD is in the representation of the execution in implementation. SPMD organizes the implementation as a single execution replicated on many processing units. While MPMD organizes explicitly the different threads of execution in the implementation. Examples of SPMD programming languages are Split-C [**Culler**], CRL [**Johnson1995**] and Composite C++ [**K.ManiChandy2005**]. Examples of MPMD programming languages are Mentat [**Grimshaw1991**], Fortran M [**Foster1995b**] and Nexus [**Foster1996**]. SPMD is close to the model presenting parallel improvements over modular programming presented in section **??**. While MPMD is closer to the programming models based on isolated process presented in the remaining of this section. The coordinations between these threads of execution were done by message passing, using PVM [**Sunderam1994**], MPI [**Snir1996**, **Walker1996**], SOAP, or the more recent REST protocols.

Theoretical Models The communication in reality are subject to various faults and attacks [**Lamport1982**] and too slow compared to execution to be synchronous. The Actor model is one of the first programming model to be explicitly designed to take these physical limitations in account [**Hewitt1977a**]. It allows to express the computation as a set of communicating actors [**Clinger1981**, 39, 38]. In reaction to a received message, an actor can create other actors, send messages, and choose how to respond to the next message. All actors are executed concurrently, and communicate asynchronously. An asynchronous communication implies that the sender continues its execution immediately after sending the message, before receiving the result of the initiated communication.

In the Actor Model, everything is an actor, even the simplest types like numbers. This level of granularity is unachievable in practice due to overhead from the asynchronous communications. Most implementations adopt a granularity on the process or function level.

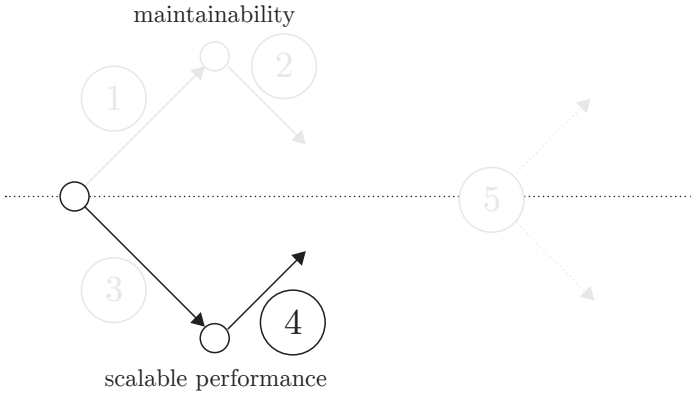
Coroutines are autonomous programs which communicate with adjacent modules as if they were input and output subroutines [19]. It is the first definition of a pipeline to implement multi-pass algorithms. Similar works include the Communicating Sequential Processes (CSP) [**Brookes1984**, 40], and the Kahn Networks [46, 47].

Table **??** presents a summary of the analysis of the paradigm presented in the previous paragraphs.

Model	Implementations	Fine level sequentiality	Coarse level message passing	→	Scalability
Event-driven programming	Node.js, Vert.X	✓	✗		✗
Lock-free Data Structures		✓	✗		✗
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✓	✗		✗
Actor Model	Scala, Akka, Play, Erlang	✓	✓		✓

Table 3.6 – Analysis of the state of the art in concurrent and parallel programming regarding performance

3.2.2 Organic Growth



The performance improvements comes directly from the industry requirements.

All these system make sens in industrial context, even the smallest. When the need for performance is higher than the need for maintainability, the research merges with the academy. If there is industrial need, there will be maintenance. The languages on the Mars Rover or in banking systems are 30 years old, and there is no community to maintain it. Yet the industry continue to maintain these languages.

However, the context of this thesis is different from a classical industrial context. During the bootstrap of a web application project, the economical context requires technologies with strong community, to pick talents from to grow the team quickly and effortlessly. It also requires these technologies to be of industrial standard, to build a reliable product. And these technology must be compatible with web technologies.

The context of this thesis requires the technology to meet all the three criteria.

- adoption by the community
- adoption by the industry
- supporting web technologies

* The field of concurrent programming is so vast it is impossible to relate here every programming languages. The previous examples are only the best known. The next focus focuses on streaming real-time applications.

review this paragraph and the transition to the next section

3.2.2.1 Exection Decomposition

The programming paradigms presented above are implemented in many existing programming languages. All major programming languages implements some form of concurrency or parallelism mechanism. The next paragraphs presents these implementations by the industry and the community. And more specifically, how they deal with the need to decompose the execution.

Event-Loop The event-loop model, featured by the DOM and Node.js with Javascript, allows concurrency but not parallelism. It decomposes the execution into sequences of callbacks functions, but keep the memory shared.

As presented in the previous section, Javascript is currently one of the most used language. This asynchronous programming model without the memory decomposition seems to be easy to develop with. It is used extensively in the community as well as in the industry. However, when the programming model requires the memory to be decompose, in order to get parallelism, it becomes more complicated to develop with, as presented in the next paragraphs.

Multi-Threading The multi-threading model allows concurrency and parallelism on certain execution region. It decomposes the execution into fork-join threads, and the memory is shared, but protected with locks. The protection of the shared memory is the reason concurrent programming is difficult to manage for most developers. Multi-threading is difficult to program with, and for this reason, it leads to poor performances. It is not heavily used in the community, where the need for concurrency is limited. In the industry, where the concurrency is often required, multi-threading is abandoned for other paradigms, such as the event loop or the actor model.

The event-loop requires an execution decomposition, but not a memory decomposition. This paradigm is heavily adopted by both the community and the industry. On the other hand, the multi-threading paradigms with locks requires an execution decomposition, and light memory decomposition. This paradigms is not heavily used in the community, and is being abandoned by the industry. This comparison between the event-loop and the multi-threading paradigms seems to indicate that the memory decomposition heavily restrains the adoption by the community. Hence, it impacts the maintainability required for the organic growth in the economical context of this thesis, as shown in table **??**.

Model	Implementations	Community adoption	Industrial adoption	Web compliant	→	Organic Growth
Event-driven programming	Node.js, Vert.X	✓	✓	✓		✓
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✗	✓	✓		✗

Table 3.7 – Analysis of the state of the art in concurrent programming regarding organic growth

The next paragraphs present solutions that forces both the execution and memory decomposition to allow parallel execution.

3.2.2.2 Actor Model

The theoretical models presented in section 3.2.1.2 are implemented in industrial languages. Example of implementation are Akka Scala and Erlang [**JoeArmstrong**].

Scala is an attempt at unifying the object model, and functional programming [**Odersky2004**]. It proposes an actor approach in its design. Akka[17] is a framework based on Scala, following the actor model to build highly scalable and resilient applications. Play[18] is a web framework based on top of Akka.

Erlang borrow the Actor model as well. It is a functional concurrent language designed by Ericsson to operate telecommunication devices [**JoeArmstrong**, **Nelson2004**]

These two example of implementation are heavily used in the industry. They are backed by strong, but small communities of passionate people, as the actor model is not trivial to understand.

Moreover, the actor model decomposes the execution into isolated parallel executions, and the memory into independent stores. These decompositions are hardly compatible with the modularity programming presented in the previous section. It is difficult for developers to manage these decompositions, executions and memory, and the modularity of the implementation. It restrain the maintainability of the implementation. Most developers are unable to manage efficiently the two decompositions required by the actor model. And novice developers seems reluctant to learn it. The next paragraphs presents some solutions based on the actor model, with the intent to mitigate the duality between execution decomposition and modularity.

Design Patterns To reduce the difficulties of the decomposition of the execution into actors, algorithmic skeletons propose predefined patterns that fit certain type of problems [**Cole1988**, **Gonzalez-Velez2010**, 20, 56]. A developer implements the problem as a specific case of a skeleton. It simplifies the communications, so that the developer can focus on its problem independently of message passing required by the distribution of execution. These solutions are hardly used by the community, but are crucial in industrial contexts. A famous example in the industry is map/reduce, introduce by Google [20].

Granularity The Service Oriented Architectures (SOA), and more recently Microservice[**Namiot201
Fowler2014**, **Namiot2014**, 25] allow developers to express an application as an assembly of services connected to each others. Some examples of frameworks are

[17] `http://akka.io/`

[18] `https://www.playframework.com/`

OSGi[19], EJB[20], Spring[21], and Seneca[22] It intends to adjust the granularity of execution decomposition to help developers to fit the two organizations, the modular organization and the parallel execution organization [**Adam2008**]. Microservices are very recent, and it is difficult to asses their usage in the community nor the industry. But they seems to be increasingly adopted, both in the industry and in the community.

3.2.2.3 Stream Processing Systems

All the solutions previously presented are designed to build general distributed systems. In the context of the web, a real-time application must process high volumes streams of requests within a certain time. Because these systems are key to business, their reliability and latency are of critical importance. Otherwise, input data may be lost or output data may lose their value. These requirements are challenging to meet in the design of such system.

Data-stream Management Systems Database Management Systems (DBMS) historically processed large volume of data, and they naturally evolved into Data-stream Management System (DSMS) to processed data streams as well. Because of this evolution, they are in rupture with imperative languages presented until now, and borrow the syntax from SQL.

DSMS concurrently run SQL-like requests on continuous data streams. The computation of these requests spread over a distributed architecture. Among the early works, we can cite NiagaraCQ [16, 60], Aurora [1, 3, 9] which evolved into Borealis [2], AQuery [**Lerner2003**], STREAM [**Arasu2003**, **Arasu2005**] and TelegraphCQ [48, 15]. More recently, we can cite DryadLINQ [43, 82], Apache Hive [**Thusoo2009**][23], Timestream [66] and Shark [**Xin2013**].

Pipeline Architecture As presented in the previous section, streaming and lazy-evaluation composition both allow a loosely coupled yet efficient composition. The pipeline architecture takes advantage of this, and composes the parallel execution in a stream, the output of one feeding the input of the next.

[19] `https://www.osgi.org/developer/specifications/`
[20] `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[21] `http://projects.spring.io/spring-framework/`
[22] `http://senecajs.org/`
[23] `https://hive.apache.org/`

SEDA is a precursor in the design of pipeline-based architecture for real-time web applications [79]. It organizes an application as a network of event-driven stages connected by explicit queues. The event-driven paradigm is similar to previous web servers implementations like Ninja and Flash [30, 63]. SEDA improves with the pipeline organization in stages.

Several projects followed and adapted the principles in this work. StreaMIT is a language to help the programming of large streaming application [76]. Storm [77] is designed by and used at Twitter to process the heavy streams of tweets. Among other works, in the industry, there are CBP [51] and S4 [62], that were designed at Yahoo, Millwheel [5] designed at Google and Naiad [**Murray2013**] designed at Microsoft.

In the litterature, there are Spidle [18], Pig Latin [**Olston2008**], Piccolo [65], Comet [37], Nectar [**Gunda2010**], SEEP [57], Legion [**Bauer2012**], Halide [**Ragan-Kelley2013**], SDG [24] and Regent [**Slaughter2015**]

At the light of this analysis, it appears that parallel programming is not heavily adopted by the community. The execution decomposition required by parallel programming improve performance scalability, but reduce the organic growth required for maintainability. Eventually, only the event-loop is a viable concurrent programming approach in the economical context of the web. And it is exactly what the numbers indicates through the heavy adoption of Javascript in the last few years.

3.2.3 Maintainability Limitations

Parallel programming requires the decomposition of memory and execution to allow the different levels of state accessibility and execution. At a fine level, the state is shared, while at a coarser level, it is isolated. Because of this decomposition, programming languages abandoned shared state, and higher-order programming between execution containers.

Indeed, the topology of the network of actors is statically defined, and the dynamical modification of the topology is mostly impossible. It is not possible to dynamically manipulate execution containers, like it is possible to manipulate functions. Therefore, higher-level programming is impossible, and limits the modularity required for maintainability. It implies to keep two mental representation of the implementation, one for the modularity, and one for the decomposition of execution. Parallel programming remains hard, and is accessible only to an elite of developers. In this regards, the memory decomposition required by parallel programming is incompatible with the economical context required in this thesis.

Model	Implementations	Community adoption	Industrial adoption	Web compliant	→	Organic Growth
Event-driven programming	Node.js, Vert.X	✓	✓	✓		✓
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✗	✓	✓		✗
Lock-free Data Structures		✗	✓	✓		✗
Actor Model	Scala, Akka, Play, Erlang	–	✓	✓		–
Skeleton	MapReduce, ...	✗	✓	✓		✗
Service Oriented Architecture	OSGi, EJB, Spring	+	✓	✓		+
Microservices	Seneca	?	?	✓		?
Data Stream System Management	DryadLINQ [43, 82],Apache Hive [**Thusoo2009**][24],Timestream [66],Shark [**Xin2013**].	✗	✓	✓		✗
Pipeline Stream Processing	SEDA, Storm, Spark Streaming,Spidle [18],Pig Latin [**Olston2008**],Piccolo [65],Comet [37],Nectar [**Gunda2010**],SEEP [57],Legion [**Bauer2012**],Halide [**Ragan-Kelley2013**],SDG [24],Regent [**Slaughter2015**].	✗	✓	✓		✗

Table 3.8 – Analysis of the state of the art in concurrent and parallel programming regarding organic growth

To fit the economical context of this thesis, a solution must provide scalability,

hence memory and execution decomposition. But at the same time proposes an abstraction for the memory decomposition, to avoid the developers to keep a double mental representation of the implementation. The next section presents some works that provides such a memory decomposition abstraction.

3.2.4 Summary

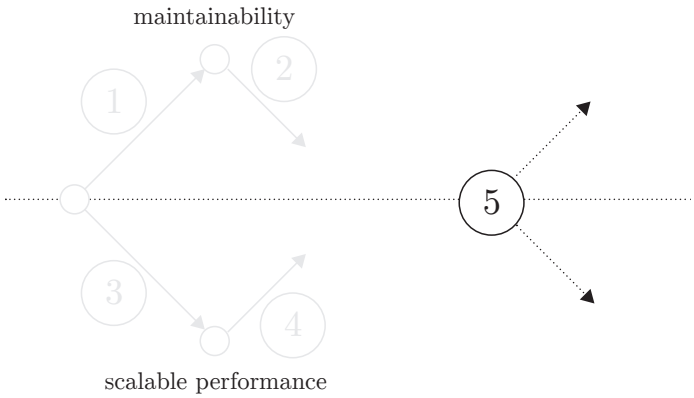
Table **??** summarizes the characteristics of the solutions presented in this section.

3.3 Memory Decomposition Abstraction

3.3.1 Runtime

The criteria to analyze the solutions presented in this section are :

- criteria 1
- criteria 2



3.3.1.1 Partitioned Global Address Space

Parallelization of the execution eventually requires the distribution of the memory. The Partitioned Global Address Space (PGAS) provides the developers with a uniform memory access on a distributed architecture. It attempts to combine

Model	Implementations	Memory decomposition abstraction	→	Maintainability
Event-driven programming	Node.js, Vert.X	✓		✓
Lock-free Data Structures		✗		✗
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✗		✗
Actor Model	Scala, Akka, Play, Erlang	✗		✗
Skeleton	MapReduce, ...	✗		✗
Service Oriented Architecture	OSGi, EJB, Spring	✗		✗
Microservices	Seneca	✗		✗
Data Stream System Management	DryadLINQ [43, 82],Apache Hive [**Thusoo2009**][25],Timestream [66],Shark [**Xin2013**].	✗		✗
Pipeline Stream Processing	SEDA, Storm, Spark Streaming,Spidle [18],Pig Latin [**Olston2008**],Piccolo [65],Comet [37],Nectar [**Gunda2010**],SEEP [57],Legion [**Bauer2012**],Halide [**Ragan-Kelley2013**],SDG [24],Regent [**Slaughter2015**].	✗		✗

Table 3.9 – Analysis of the state of the art regarding maintainability

Model	Implementations	Maintainability	Organic Growth	Scalability
Event-driven programming	Node.js, Vert.X	✓	✓	✗
Lock-free Data Structures		✗	✓	✓
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✗	✓	✓
Actor Model	Scala, Akka, Play, Erlang	✗	–	✓
Skeleton	MapReduce, ...	✗	✗	✓
Service Oriented Architecture	OSGi, EJB, Spring	✗	+	✓
Microservices	Seneca	✗	✗	✓
Data Stream System Management	DryadLINQ [43, 82],Apache Hive [**Thusoo2009**][26],Timestream [66],Shark [**Xin2013**].	✗	✗	✓
Pipeline Stream Processing	SEDA, Storm, Spark Streaming,Spidle [18],Pig Latin [**Olston2008**],Piccolo [65],Comet [37],Nectar [**Gunda2010**],SEEP [57],Legion [**Bauer2012**],Halide [**Ragan-Kelley2013**],SDG [24],Regent [**Slaughter2015**].	✗	✗	✓

Table 3.10 – Summary of the analysis of the state of the art in concurrent and parallel programming

the advantage of SPMD programming style for distributed memory systems, with the data referencing semantics of shared memory systems. Each computing node executes the same program, and provide its local memory to be shared with all the other nodes. The PGAS programming model assure the remote accesses and synchronization of memory across nodes, and enforces locality of reference, to reduce the communication overhead. Examples of implementation of the PGAS model

are CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**] and HPX [**Kaiser2015**]*.

⚠ include as well HPX: A Task Based Programming Model in a Global Address Space by Kaiser, Heller, Adelstein-Lelbach

3.3.1.2 Dynamic Distribution of Execution

An interesting work following SEDA, is Leda [70, 71].It follows the PCAM design methodology [**Foster1995**] to propose a model where the stages of the pipeline are defined only by their role in the application. The actual execution distribution in stages is defined automatically, only after the development, during deployment. This automation blurs the distinction between the parallel organization of execution, and the modular organization of implementation. It manages the execution organizations to helps the developer focus on the modular organization.

3.3.2 Compilation

"It is a mistake to attempt high concurrency without help from the compiler"
— R. Behren, J. Condit, E. Brewer [11]

.

D. Parnas advocates the use of an assembler to conciliate the two approaches [64].

When showing the incompatibility between the two organization, D. Parnas advocated conciliating the two methods using an assembler to transform the development organization into the execution organization [64]. This section presents the state of the art to extract parallelization from sequential programs through code transformation and compilation.

*

⚠ read and include [**Catanzaro2009**]

3.3.2.1 Parallelism Extraction

As the only requirement to parallelism is the commutativity of operations [69, 17], a compiler needs to identify the commutative operations transform a sequential program so as to parallelize its execution [69].

An important work was done to parallelize loop iterations [**Mauras1989**, **Chen2008**, 6, 10, 67], particularly using the polyhedral compilation method [**Yuki2013**, **Grosser2011**, **Trifunovic2010**, **Bastoul2004**]. However, this data parallelism is limited to scientific applications because of their heavy use of loops on matrices and vectors. The

performance gains are limited in common sequential programs, as the execution remains sequential outside of loops [7, 17].

To improve performance gains further, some compilers identify the data-flow inside sequential programs to allow parallelism on the whole program, and not only on its loops [**Beck1991**, **Catanzaro2009**, **Li2012**]. Moreover, the data-flow representation and execution of a program is well suited for modern data processing applications [24], as well as web services [70]. *

However, the limitation of modular programming regarding parallelization persists. In a purely functional language with immutability, higher-order functions are referentially transparent which implies commutativity hence parallelism *. However, in a functional language with mutable data, closures remains a challenge to parallelize, because of the memory references shared across the program [**Harrison1989**, **Nicolay2010**, 55]. The next two paragraphs presents two directions to improve the state of the art in parallel compilation. The first paragraph presents static analysis, while the second presents annotations systems.

⚠ TODO Extract parallelism compilers from these : Load balanced pipeline parallelism [**Kamruzzaman2013**], Regent [**Slaughter2015**], Cilk-P, On-the-Fly Pipeline Parallelism[**Lee2013**]

3.3.2.2 Static analysis

⚠ Add reference of parallel purely functional languages

Compilers analyze the control-flow of a program to detect the side-effects causing dependencies between statements [**Allen1970**]. The point-to analysis, presented by L. Andersen [8] is a popular approach to identify these side-effects in the memory representation. The points-to-analysis was adapted for Javascript [44, 73, 78], and is a useful tool to analyze a program. However, this analysis is not sufficient to track the dynamic control-flow of higher-order functions [**Shivers1991**] like used in Javascript.

The Operational Semantics is an example of abstract interpretation technique that allows to statically reason on the behavior of programs[52, 72, 28, 27, 13]. * Abstract interpretation techniques are more adapted for program with higher-order functions, and are successfully used for security applications [**Chudnov2015**, **Dolby2015**, 42, 45, 81, 53]*. * *

⚠ TODO review this paragraph

⚠ Update the citation for Dolby2015

However, static analysis techniques are too imprecise, and expensive for the performance gain to be profitable in languages as dynamic as Javascript. Instead, some compilers relies on annotations from the developers.

3.3.2.3 Annotations

⚠ Read and include [35]

Extracting parallel dataflow from an imperative, sequential implementation is a hard problem [**Johnston2004a**]. Some works proposed to rely on annotations

⚠ Read and include Effective race detection for event-driven programs, by Raychev, Vechev, Sridharan

from the developer to help the identify the possible side-effects between operations [**Vandierendonck2010a**, 24].

Many compilers rely on annotations from the developer to build highly parallel executables. Such annotations are especially relevant for accelerators such as GPUs or FPGAs, because the development effort yield huge performance improvements. Examples of such compilers are OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**].

* *

read and include [**Catanzaro2009**]

read and include Accelerators: Using data parallelism to program gpus for general purpose uses

However, the burden of detecting commutativity of operations, or independence of operations fall back to the developer. In this regard, these solutions successfully improve performances, but are unable to fix the rupture between performance and maintainability. These solutions are indeed very close to the performance oriented solutions presented in the section 3.2.

3.3.2.4 Compilation Limitations

The static analysis of static, low level languages like FORTRAN or C, brings performance improvements. However for more dynamic, higher-level languages like Javascript, the static analysis is not sufficient to identify correctly the dependencies between operations to parallelize them. And parallel compilers often fall back on relying on annotation provided by developers. So, in this regards, it seems that the accessibility of development gained by higher-level programming is detrimental to performance.

3.3.3 Organic Growth

A system is maintainable only if there is competences available to maintain it.

The criteria to analyze the solutions presented in this section regarding the organic growth are :

- adoption by the community
- adoption by the industry
- supporting web technologies

The first two criteria make sure that the technology is growing organically with a passionate community, and backed by industrial needs. The last criteria assures the fitting of the technologies with our economical context of a web application.

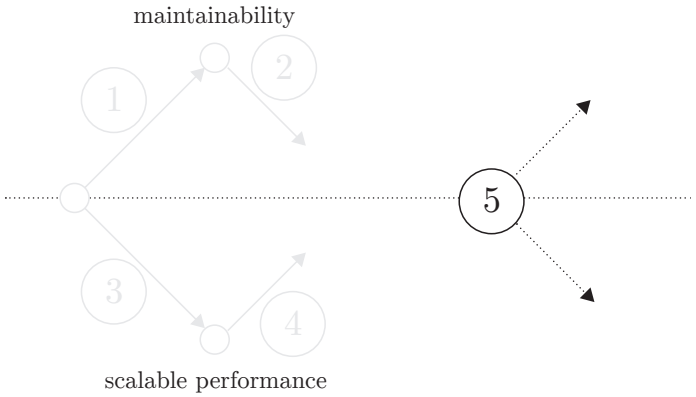
Model	Implementations	Memory decomposition abstraction	→	Maintainability
PGAS	CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**], HPX [**Kaiser2015**]	✓		✓
Dynamic distribution	LEDA	✓		✓
Parallelism extraction	Polyhedral compiler	✓		✓
Annotations	OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**]	✓		✓

Table 3.11 – Analysis of the state of the art regarding maintainability

3.3.4 Limitations

To allow a continuous development of a web service, a language must present these three features :

- scalability
- maintainability
- abstract the decomposition



The first allows the evolution of the performance. The second assure the evolution of the implementation. The third is required to allow these two first features to coexist.

3.3.5 Summary

3.4 Analysis

Model	Implementations	Modularity	Organic Growth	Scalability
Software maintainability				
Imperative Programming	C	✗	✗	✗
Object-Oriented Programming	C++, Java	✗	✓	✗
Functional Programming	Scheme, Miranda, Haskell	✓	✗	✗
Dynamic Scripting	Javacript, Python, Ruby, Go	✓	✓	✗

Concurrent Programming				
Event-driven programming	Node.js, Vert.X	✓	✓	✗
Multi-threading programming	Lock, Mutex, Semaphores, Guarded regions	✗	✓	✓
Lock-free Data Structures		✗	✓	✓
Parallel Programming				
SPMD	Split-C, CRL, Composite C++	✗	✗	✓
MPMD	Mentat, Fortran M, Nexus	✗	✗	✓
Actor Model	Scala, Akka, Play, Erlang	✗	✗	✓
Data Stream System Management	DryadLINQ [43, 82],Apache Hive [**Thusoo2009**][27],Timestream [66],Shark [**Xin2013**].	✗	✗	✓
Pipeline Stream Processing	SEDA, Storm, Spark Streaming,Spidle [18],Pig Latin [**Olston2008**],Piccolo [65],Comet [37],Nectar [**Gunda2010**],SEEP [57],Legion [**Bauer2012**],Halide [**Ragan-Kelley2013**],SDG [24],Regent [**Slaughter2015**].	✗	✗	✓
Decomposition Abstraction				

[27] `https://hive.apache.org/`

PGAS	CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**], HPX [**Kaiser2015**]	✓	✗	✓
Dynamic distribution	LEDA	✓	✗	✓
Parallelism extraction	Polyhedral compiler	✓	✗	✓
Annotations	OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**]	✓	✗	✓

TODO

Model	Implementations	Memory decomposition abstraction	→	Maintainability
PGAS	CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**], HPX [**Kaiser2015**]	✓		✓
Dynamic distribution	LEDA	✓		✓
Parallelism extraction	Polyhedral compiler	✓		✓
Annotations	OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**]	✓		✓

Table 3.12 – Analysis of the state of the art regarding maintainability

Model	Implementations	Memory decomposition abstraction	→	Maintainability
PGAS	CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**], HPX [**Kaiser2015**]	✓		✓
Dynamic distribution	LEDA	✓		✓
Parallelism extraction	Polyhedral compiler	✓		✓
Annotations	OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**]	✓		✓

Table 3.13 – Analysis of the state of the art regarding maintainability

Model	Implementations	Modularity	Organic Growth	Scalability
PGAS	CoArray Fortran [**Numrich1998**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], Chapel[**Chamberlain2007**], OpenSHMEM [**Chapman2010**]. Kokko [**Edwards2012**], UPC++ [**Zheng2014**], RAJA [**Hornung2014**], ACPdl [**Ajima2015**], HPX [**Kaiser2015**]	✓	✗	✓
Dynamic distribution	LEDA	✓	✗	✓
Parallelism extraction	Polyhedral compiler	✓	✗	✓
Annotations	OpenMP [**Dagum1998**], OpenCL [**Stone2010**], CUDA [**Nvidia2007**] Cg [54], Brook [14], Liquid Metal [**Huang2008**]	✓	✗	✓

Table 3.14 – Summary of the analysis of the state of the art

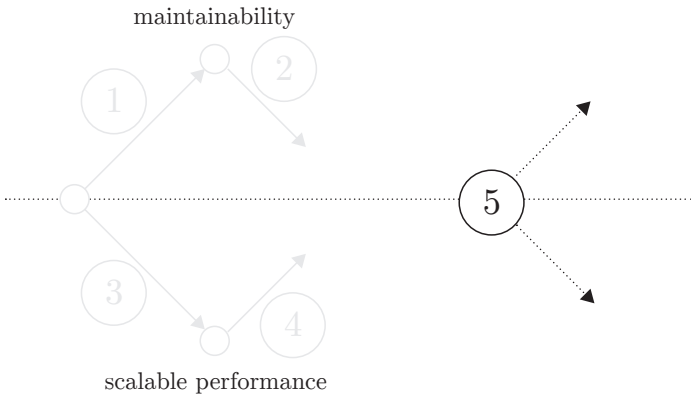
Chapter 4

Pipeline parallelism for Javascript

4.1 Seamless Development

*

TODO title not clear enough



*

*

read and include [Catanzaro2009] it is about Productivity language JIT compilation into efficient language And get all the paper that cite this one.

The section 3.1 shows that the modular organization enabled by functional programming is the best way to improve maintainability. But it requires the use of a global memory store which conflicts with performance. Compilation is a solution to reduce this conflict, but is not yet satisfactory enough for high performance scalability. On the other hand, the section 3.2 shows that to attain performance scalability, an application needs to multiply the exclusive accesses to its state. That implies

selective JIT : Dcg: an efficient retargetable dynamic code generation system, by Engler and Proebsting

follow a distributed organization of its state to provide isolation and immutability, which negatively impacts modularity, hence maintainability. Some works provide a uniform memory access to improve maintainability, despite the distributed execution.

The evolution of the economical constraints of a web application requires to repeatedly switch between maintainability and performance scalability. The incompatibility between the two organizations implies technological ruptures at each switch. Huge developing efforts are pulled to translate manually from one organization into the other, and later to maintain the implementation despites its unmaintainable nature. There is still room for improvements on a compromise between maintainability and performance scalability.

The state of the art highlighted that

- maintainability requires lazy-evaluation and higher-order programming, section 3.1.1.2, and
- higher-order programming requires a global memory abstraction, section **??**,

Javascript is a functional language that features higher-order programming and a global memory abstraction. Moreover, node.js features a streaming approach with the event-loop execution model, similar to the lazy evaluation. These reasons make Javascript a language of choice for developing web application.

And that

- scalable performance requires parallelism, and
- parallelism requires exclusive accesses on the state through isolation and immutability.

Eventually, web development is heading toward a streaming approach with pipeline processing.

*

TODO dependency schema of these highlights

This thesis proposes an equivalence between the global memory and control flow on one hand, and memory isolation with message passing on the other hand. It proposes this equivalence as a solution to conciliate the scalable performance and maintainability. As explained below, the concurrency model of the event-loop execution model, and the parallel approach of the pipeline execution model are very similar. The goal of this thesis is to allow to compile one execution model into the other, to allow developers to constantly keep two organization of their implementation, allowing them to focus on both maintainability and scalable performance.

4.1.1 Equivalence

The next paragraphs introduces this equivalence between the event-loop execution model and the pipeline execution model. The equivalence addresses two *levels**, as illustrated in figure 4.1, the control flow, and the memory isolation.

not the good word

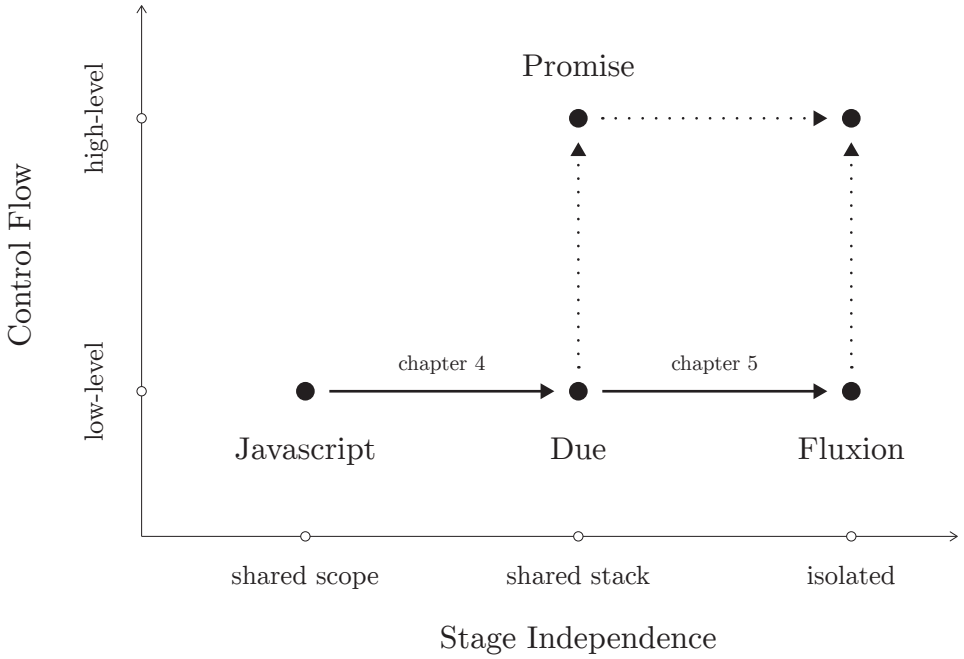


Figure 4.1 – Roadmap

4.1.1.1 Rupture Point

The execution of the pipeline architecture is well delimited in isolated stages. Each stage has its own thread of execution, and is independent from the others. On the contrary, the code of the event-loop is linear because of the continuation passing style and the common memory store. However, the execution of the different callbacks are as distinct as the execution of the different stages of a pipeline. The call stacks of two callbacks are distinct. Therefore, an asynchronous function call represents the rupture between two call stacks. It is a rupture point, and is equivalent to a data stream between two stages in the pipeline architecture.

Both the pipeline architecture and the event-loop present these rupture points. The detection of rupture points allows to map a pipeline architecture onto the implementation following the event-loop model. To allow the transformation from one to

the other, this thesis studies the possibility to detect rupture points, and to distribute the global memory into the parts defined by these rupture points. The detection of rupture points is addressed in chapter 4.

It presents the extraction of a pipeline of operations from a Javascript application. Indeed, such pipeline is similar to the one exposed by Promises. The chapter proposes a simpler alternative to the latter called Dues. However, these operations still require a global memory for coordination so they are not executed in parallel.

4.1.1.2 Invariance

The transformation should preserve the invariance as expressed by the developer to assure the correctness of the execution. The partial ordering of events in a system, by opposition to total ordering, is sufficient to assure this correctness. The global memory is a way to assure the total ordering of events, and the message passing coordination is a way to assure partial ordering of events. Therefore, to assure the correctness of the execution of a system, the state coordination with a global memory is equivalent to message passing coordination. And it is possible, at least for some rupture points, to transform the global memory coordination into message passing while conserving the correctness of execution.

In order to preserve the invariance assured by the event-loop model after the transformation, each stage of the pipeline needs to have an exclusive access to memory. The global memory needs not to be split into parts and distributed into each of the stages. To assure the missing coordinations assured by the shared memory between the stages, the transformation should provide equivalent coordination with message passing. The isolation and replacement of the global memory is fully address in chapter 5, with the introduction of isolated containers called Fluxions.

From here, the reader should be confortable with the event-loop, and the analogy we drawn between the event-loop and a pipeline. The problematic is now clear : how to split the heap so that each asynchronous callback has its own exclusive heap ?

4.2 Callback identification

4.2.1 TODO

4.3 Callback isolation

We explain in this section the compilation process we developped to isolate the memory access for each callbacks. The result of this process should be two-fold.

First each callback should have an exclusive access on a region of the memory. So that two different callback can be executed in parallel. And it should be clear for each callback, what are the variable needed from upstream callbacks, and what are the variable to send downstream.

4.3.1 Propagation of variables

4.3.1.1 Scope identification

In section **??**, we explained that Javascript is roughly lexically scoped. A consequence is that the declaration of contexts can be inferred statically. For example, in a lexically scoped, strongly typed, compiled language, the compiler know the content of each scope during compile time, and can prepare the memory stack to store the variables in each scope.

In most languages, the memory is in two parts : the stack, and the heap. The stack is statically scoped, and its layout is known at compile time. The heap, on the other hand is dynamically allocated. Its layout is built at run time.

But Javascript is a dynamic language, perhaps the most dynamic of all languages. It doesn't have this distinction between stack and heap. Every variable is dynamically allocated on the heap. That induce two consequences. The first is that Javascript provides two statements to dynamically modify the lexial scope : `eval` and `with`. The second is that to know the layout of the heap, we need to use static analysis tools. In the next two sections, we adress these two consequences.

4.3.1.2 Break the lexical scope

Without these statements, `eval` and `with`, Javascript is lexically scoped. It is possible to infer the scope of each variable at compile time.

The `with` statement continue the execution using an expression as the lexical scope. As the provided expression is dynamically evaluated, it is possible to dynamically modify the lexical scope. The code snippet below show an example of such a situation.

```
1 var aliveCat = {isAlive: true};
2 var deadCat = {isDead: true}
3
4 with (Math.random() > 0.5 ? aliveCat : deadCat) {
5   isAlive;
6   // Half the time -> ReferenceError: isAlive is not defined
7   // Half the time -> true;
8 }
```

The variable `isAlive` is defined only in the object `aliveCat`. The presence of the variable `isAlive` in the lexical environment within the `with` statement cannot be determined statically, as the lexical environment is dynamically linked to either `aliveCat` or `deadCat`.

Note that the MDN reference page on `with`[1] says that *Using* `with` *is not recommended, and is forbidden in ECMAScript 5 strict mode.*

The

Not to be mistaken with the `this` operator. It is possible to dynamically change the content of an object,

```
function stuff() {
  this.x = 42;
}

stuff.call({})
```

However, even if Javascript is lexically scoped, the memory is still dynamicall allocated and manipualeted, so that it is not possible to actually infer the memory layout at compiler time only with lexical scope analysis, and without deeper static analysis.

4.3.1.3 Scope Leaking

To infer the layout of the heap at compile time, static analysis tools are used, like the points-to analysis, developped by Andersen in its PhD thesis [8]. For such analysis, the memory is splitted at the access scale. In low-level languages, like C/C++, the memory is mainly managed by the developer. Allowing access to the memory at a small grained scale : up to the address. It impose the analysis to split the memory to the adress scale in some cases. In higher-level languages, like Javascript, the developer cannot access the memory to the adress scale. The memory is accessed at a coarser scale : the property scale. (At the exception of some arrays and buffers, that mimic, and are mapped to actual memory adresses for performance reasons.)

4.3.1.4 Propagation of execution and variables

For the execution of each callback / stage, the corresponding part of the state is local, and the rest is remote, and inaccessible. We are going to explain why it must remain inaccessible.

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`

While a callback is executing a request, the previous callback (the up stream callback) is executing the next request. The next request will arrive at the current callback some time in the future. The modification done in the state of the upstream callback will propagate only later in the current callback. The state of the upstream callback is in a different time frame than the state of the current callback.

To really understand that, we need to compare this execution with the execution on a unique event-loop. If the current callback executes, then the upstream callback might have, or might not have started to execute the next request. But as soon as the current callback executes, the modifications done on the states, are immediatly propagated, so that the upstream callback can take them into account for the next request.

However, if the two callbacks are distant, then the modification of the current callback will not immediatly propagate to the upstream callback. During the propagation, the upstream callback might execute requests than would not be aware of the state modification from the current callback (from downstream). That is why we say the upstream callback and the current callback are in two different time frame. Propagating the state modification upstream is like going backward in time, it is impossible. That is why the execution, and the state modification propagation must always flow downstream.

As a note, I must add that if an upstream and a downstream callbacks are on the same event-loop, then this doesn't apply. it is like a loop in the time : the modification immediatly propagate from downstream to upstream.

Chapter 5

Pipeline extraction

The previous chapter presented globally the state of the art in designing systems to scale in performance, and in maintenance. It refined the scope of this thesis to the study of the opposition between maintenance scalability and performance scalability in streaming web applications. It concluded with the objectives of this thesis, which is to find an equivalence between the two opposed organizations. The maintenance scalability organization, supported by modular programming, higher-order programming and a global memory store. The performance scalability organization, supported by the parallelism of memory and exuction distribution. The equivalence between these two organization is in two steps, as presented in figure 4.1. This chapter presents the first step in this equivalence. That is to identify and extract a pipeline of execution inside an application following the first organization. In this work, we focus on Javascript, and specifically node.js applications. In this chapter, I define further the higher-order programming concepts.

In Javascript, functions are first-class citizens ; it allows to manipulate them like any object, and to link them to react to asynchronous events, *e.g.* user inputs and remote requests. These asynchronously triggered functions are named callbacks, and allow to efficiently cope with the distributed and inherently asynchronous architecture of the Internet. To execute a suite of asynchronous functions, callbacks are nested one into the other. This nesting, if not organized properly, can result in unreadable layer of callbacks, commonly presented as *callback hell*[1], or *pyramid of doom*.

Promises are another way to organize a suite of asynchronous operations avoiding this callback hell. They organize the operations as a well-defined pipeline. Moreover, Promises provide additional control over the asynchronous execution flow, than call-

[1] `http://maxogden.github.io/callback-hell/`

backs. They are part of the next version of the Javascript language, ECMAScript 6[2]. To avoid the equivalence being unnecessarily incomplete, we present an alternative to Promise, called Due. Due organize the operations like Promises, as a well-defined pipeline, while discarding the unnecessary additional control over the asynchronous flow.

This chapter present an equivalence, and a compiler to identify the pipeline of operating underlying in a Javascript application using callbacks, and extract it to express it as Dues. This compiler has been tested over 64 *Node.js* packages from the node package manager (npm[3]). 55 packages were incompatible with the compiler, 9 packages were compiled with success.

Callbacks, Promises and Dues are further defined in section 5.1. Section 5.2 explains the transformation from imbrications of callbacks to sequences of Dues. Section 6.2 presents a compiler to automate the application of this equivalence. And finally, the developed compiler is evaluated in section 6.3.

5.1 Definitions

5.1.1 Callback

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.

As we will see later, Promises are designed as placeholders for a unique outcome. Iterators and Listeners are invoked multiple times resulting in multiple outcomes. Only continuations are equivalent to Promises. Therefore, we focus on continuations in this paper.

Callbacks are often mistaken for continuations; the former are not inherently asynchronous while the latter are. In a synchronous paradigm, the sequentiality of the execution flow is trivial. An operation needs to complete before executing the

[2] `http://people.mozilla.org/~jorendorff/es6-draft.html`
[3] `https://www.npmjs.com/`

next one. In an asynchronous paradigm, parallelism is trivial, but the sequentiality of operations needs to be explicit. Continuations are the functional way of providing this control over the sequentiality of the asynchronous execution flow.

A continuation is a function passed as an argument to allow the callee not to block the caller until its completion. The caller is able to continue the execution while the callee runs in background. The continuation is invoked later, at the termination of the callee to continue the execution as soon as possible and process the result; hence the name continuation. It provides a necessary control over the asynchronous execution flow. It also brings a control over the data flow which essentially replaces the `return` statement at the end of a synchronous function. At its invocation, the continuation retrieves both the execution flow and the result.

The convention on how to hand back the result must be common for both the callee and the continuation. For example, in *Node.js*, the signature of a continuation uses the *error-first* convention. The first argument contains an error or `null` if no error occurred; then follows the result. Listing 5.1 is a pattern of such a continuation. However, continuations don't impose any conventions; indeed, other conventions are used in the browser.

```
my_fn(input, function continuation(error, result) {
  if (!error) {
    console.log(result);
  } else {
    throw error;
  }
});
```

Listing 5.1 – Example of a continuation

The callback hell occurs when many asynchronous calls are arranged to be executed sequentially. Each consecutive operation adds an indentation level, because it is nested inside the continuation of the previous operation. It produces an imbrication of calls and function definitions, as shown in listing 5.2. We say that continuations lack the chained composition of multiple asynchronous operations. Promises allow to arrange such a sequence of asynchronous operations in a more concise and readable way.

```
my_fn_1(input, function cont(error, result) {
  if (!error) {
    my_fn_2(result, function cont(error, result) {
      if (!error) {
        my_fn_3(result, function cont(error, result) {
          if (!error) {
            console.log(result);
          } else {
            throw error;
          }
        });
```

```
12        } else {
13          throw error;
14        }
15      });
16    } else {
17      throw error;
18    }
19  });
```

Listing 5.2 – Example of a sequence of continuations

5.1.2 Promise

In a synchronous paradigm, the sequentiality of the execution flow is trivial. While in an asynchronous paradigm, this control is provided by continuations. Promises provide a unified control over the execution flow for both paradigms. The ECMAScript 6 specification[4] defines a Promise as an object that is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. Promises expose a `then` method which expects a continuation to continue with the result; this result being synchronously or asynchronously available.

Promises force another control over the execution flow. According to the outcome of the operation, they call one function to continue the execution with the result, or another to handle errors. This conditional execution is indivisible from the Promise structure. As a result, Promises impose a convention on how to hand back the outcome of the deferred computation, while classic continuations leave this conditional execution to the developer.

```
1 var promise = my_fn_pr(input)
2
3 promise.then(function onSuccess(result) {
4   console.log(result);
5 }, function onError(error) {
6   throw error;
7 });
```

Listing 5.3 – Example of a promise

Promises are designed to fill the lack of chained composition from continuations. They allow to arrange successions of asynchronous operations as a chain of continuations, by opposition to the imbrication of continuations illustrated in listing 5.2. That is to arrange them, one operation after the other, in the same indentation level.

The listing 5.4 illustrates this chained composition. The functions `my_fn_pr_2` and `my_fn_pr_3` return promises when they are executed, asynchronously. Because

[4] `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects`

these promises are not available synchronously, the method `then` synchronously returns intermediary Promises. The latter resolve only when the former resolve. This behavior allows to arrange the continuations as a flat chain of calls, instead of an imbrication of continuations.

```
my_fn_pr_1(input)
.then(my_fn_pr_2, onError)
.then(my_fn_pr_3, onError)
.then(console.log, onError);

function onError(error) {
  throw error;
}
```

Listing 5.4 – A chain of Promises is more concise than an imbrication of continuations

The Promises syntax is more concise, and also more readable because it is closer to the familiar synchronous paradigm. Indeed, Promises allow to arrange both the synchronous and asynchronous execution flow with the same syntax. It allows to easily arrange the execution flow in parallel or in sequence according to the required causality. This control over the execution leads to a modification of the control over the data flow. Programmers are encouraged to arrange the computation as series of coarse-grained steps to carry over inputs. In this sense, Promises are comparable to some coarse-grained data-flow programming paradigms, such as Flow-based programming [59].

5.1.3 From continuations to Promises

As detailed in the previous sections, continuations provide the control over the sequentiality of the asynchronous execution flow. Promises improve this control to allow chained compositions, and unify the syntax for the synchronous and asynchronous paradigm. This chained composition brings a greater clarity and expressiveness to source codes. At the light of these insights, it makes sense for a developer to switch from continuations to Promises. However, the refactoring of existing code bases might be an operation impossible to carry manually within reasonable time. We want to automatically transform an imbrication of continuations into a chained composition of Promises.

We identify two steps in this transformation. The first is to provide an equivalence between a continuation and a Promise. The second is the composition of this equivalence. Both steps are required to transform imbrications of continuations into chains of Promises.

Because Promises bring chained composition, the first step might seem trivial as it does not imply any imbrication to transform into chain. However, as explained in

section 5.1.2, Promises impose a control over the execution flow that continuations leave free. This control induces a common convention to hand back the outcome to the continuation.

In the Javascript landscape, there is no dominant convention for handing back outcomes to continuations. In the browser, many conventions coexist. For example, *jQuery*'s `ajax`[5] method expects an object with different continuations for success, errors and various other events during the asynchronous operation. Q[6], a popular library to control the asynchronous flow, exposes two methods to define continuations: `then` for successes, and `catch` for errors. On the other hand, the *Node.js* API always used the *error-first* convention, encouraging developers to provide libraries using the same convention. In this large ecosystem the *error-first* convention is predominant. All these examples use different conventions than the Promise specification detailed in section 5.1.2. They present strong semantic differences, despite small syntactic differences.

To translate these different conventions into the Promises one, the compiler would need to identify them. Such an identification might be possible with static analysis methods such as the points-to analysis [78], or a program logic [27, 13]. However, it seems impracticable because of the number and semantical heterogeneity of these conventions. Indeed, in the browser, each library seems to provide its own convention.

In this paper, we are interested in the transformation from imbrications to chains, not from one convention to another. The *error-first* convention, used in *Node.js*, is likely to represent a large, coherent code base to test the equivalence. Indeed contains currently more than 125 000 packages. For this reason, we focus only on the *error-first* convention. Thus, our compiler is only able to compile code that follows this convention. The convention used by Promises is incompatible. We propose an alternative specification to Promise following the *error-first* convention. In the next section we present this specification called Due.

The choice to focus on *Node.js* is also motivated by our intention to compare later the chained sequentiality of Promises with the data-flow paradigm. *Node.js* allows to manipulate streams of messages. This proved to be efficient for real-time web applications manipulating streams of user requests. Both Promises and data-flow arrange the computation in chains of independent operations.

[5] `http://api.jquery.com/jquery.ajax/`
[6] `http://documentup.com/kriskowal/q/`

5.1.4 Due

A Due is an object used as placeholder for the eventual outcome of a deferred operation. Dues are a simplification of the Promise specification. They are essentially similar to Promises, except for the convention to hand back outcomes. They use the *error-first* convention, like *Node.js*, as illustrated in listing 5.5. The implementation of Dues and its tests are available online[7]. A more in-depth description of Dues and their creation follows in the next paragraphs.

```
1  var my_fn_due = require('due').mock(my_fn);
2
3  var due = my_fn_due(input);
4
5  due.then(function continuation(error, result) {
6    if (!error) {
7      console.log(result);
8    } else {
9      throw error;
10   }
11 });
```

Listing 5.5 – Example of a due

A due is typically created inside the function which returns it. In listing 5.5, line 1, the `mock` method wraps `my_fn` in a Due-compatible function. The rest of this code is similar to the Promise example, listing 5.3.

We illustrate in listing 5.6 the creation of a Due through the `mock` method. At its creation, line 6, the Due expects a callback containing the deferred operation, which is `my_fn` here. This callback is executed synchronously with the function `settle` as argument to settle the Due, synchronously or asynchronously. The `settle` function is pushed at the end of the list of arguments. The callback invokes the deferred operation with this list of arguments, and the current context, line 8. When finished, the latter calls `settle` to settle the Due and save the outcome. Settled or not, the created Due is always synchronously returned. Its `then` method allows to define a continuation to retrieve the saved outcome, and continue the execution after its settlement. If the deferred operation is synchronous, the Due settles during its creation and the `then` method immediately calls this continuation. If the deferred operation is asynchronous, this continuation is called during the Due settlement.

```
1  Due.mock = function(my_fn) {
2    return function mocked_fn() {
3      var _args = Array.prototype.slice.call(arguments),
4          _this = this;
5
6      return new Due(function(settle) {
7        _args.push(settle);
```

[7] `https://www.npmjs.com/package/due`

```
8         my_fn.apply(_this, _args);
9       })
10    }
11  }
```

Listing 5.6 – Creation of a due

The composition of Dues is the same than for Promises (see section 5.1.2). Through this chained composition, Dues arrange the execution flow as a sequence of actions to carry on inputs.

This simplified specification adopts the same convention than *Node.js* for continuations to hand back outcomes. Therefore, the equivalence between a continuation and a Due is trivial. Dues are admittedly tailored for this paper, hence, they are not designed to be written by developers, like Promises are. They are an intermediary step between classical continuations and Promises. We present in section 5.2 the equivalence between continuations and Dues.

5.2 Equivalence

*

TODO this title is not clear

We present the transformation from a nested imbrication of continuations into a chain of Dues. We explain the three limitations imposed by our compiler for this transformation to preserve the semantic. They preserve the execution order, the execution linearity and the scopes of the variables used in the operations.

5.2.1 Execution order

Our compiler spots function calls with a continuation, which are similar to the abstraction in (5.1). It wraps the function fn into the function $fn_{\mathbf{due}}$ to return a Due. And it relocates the continuation in a call to the method **then**, which references the Due previously returned. The result should be similar to (5.2). The differences are highlighted in bold font.

$$fn([arguments], continuation) \tag{5.1}$$

$$fn_{\mathbf{due}}([arguments])\mathbf{.then}(continuation) \tag{5.2}$$

The execution order is different whether *continuation* is called synchronously, or asynchronously. If fn is synchronous, it calls the *continuation* within its execution. It might execute *statements* after executing *continuation*, before returning. If fn is

asynchronous, the continuation is called after the end of the current execution, after fn. The transformation erases this difference in the execution order. In both cases, the transformation relocates the execution of *continuation* after the execution of fn. For synchronous fn, the execution order changes ; the execution of *statements* at the end of fn and the continuation switch. The latter must be asynchronous to preserve the execution order.

5.2.2 Execution linearity

Our compiler transforms a nested imbrication of continuations, which is similar to the abstraction in (5.3) into a flatten chain of calls encapsulating them, like in (5.4).

$$
\begin{aligned}
&fn1([arguments], cont1\{\\
&\quad declare\ variable \leftarrow result\\
&\quad fn2([arguments], cont2\{\\
&\qquad print\ variable\\
&\quad \})\\
&\})
\end{aligned}
\tag{5.3}
$$

$$
\begin{aligned}
&\textbf{declare variable}\\
&fn1_{\textbf{due}}([arguments])\\
&\textbf{.then}(cont1\{\\
&\quad variable \leftarrow result\\
&\quad fn2_{\textbf{due}}([arguments])\\
&\})\\
&\textbf{.then}(cont2\{\\
&\quad print\ variable\\
&\})
\end{aligned}
\tag{5.4}
$$

An imbrication of continuations must not contain any loop, nor function definition that is not a continuation. Both modify the linearity of the execution flow which is required for the equivalence to keep the semantic. A call nested inside a loop returns multiple Dues, while only one is returned to continue the chain. A function definition breaks the execution linearity. It prevent the nested call to return the Due expected to continue the chain. On the other hand, conditional branching leaves the execution linearity and the semantic intact. If the nested asynchronous function is not called, the execution of the chain stops as expected.

5.2.3 Variable scope

In (5.3), the definitions of *cont*1 and *cont*2 are overlapping. The *variable* declared in *cont*1 is accessible in *cont*2 to be printed. In (5.4), however, definitions of *cont*1 and *cont*2 are not overlapping, they are siblings. The *variable* is not accessible to *cont*2. It must be relocated in a parent function to be accessible by both *cont*1 and *cont*2. To detect such variables, the compiler must infer their scope statically. Languages with a lexical scope define the scope of a variable statically. Most imperative languages present a lexical scope, like C/C++, Python, Ruby or Java. The subset of Javascript excluding the built-in functions `with` and `eval` is also lexically scoped. To compile Javascript, the compiler must exclude programs using these two statements.

5.3 Compiler

We build a compiler to automate the application of this equivalence on existing Javascript projects. The compilation process contains two important steps, the identification of the continuations, and the generation of chains.

5.3.1 Identification of continuations

The first compilation step is to identify the continuations and their imbrications. The nested imbrication of callbacks only occurs when they are defined *in situ*. The compiler detects a function definition within the arguments of a function call. This detection is based on the syntax, and is trivial.

Not all detected callbacks are continuations, but the equivalence is applicable only on the latter. A continuation is a callback invoked only once, asynchronously. Spotting a continuation implies to identify these two conditions. There is no syntactical difference between a synchronous and an asynchronous callee. And it is impossible to assure a callback to be invoked only once, because the implementation of the callee is often statically unavailable. Therefore, the identification of continuations is necessarily based on semantical differences. To recognize these differences, the compiler would need to have a deep understanding of the control and data flows of the program. Because of the highly dynamic nature of Javascript, this understanding is either unsound, limited, or complex. Instead, we choose to leave to the developer the identification of compatible continuations among the identified callbacks. They are expected to understand the limitations of this compiler, and the semantic of the code to compile.

We provide a simple interface for developers to interact with the compiler. We built this interface around the compiler in a web page available online[8] to reproduce the tests. The web technologies allow to quickly build an interface for a wide variety of computing devices.

This interaction prevents the complete automation of the individual compilation process. However, we are working on an automation at a global scale. We expect to be able to identify a continuation only based on the name of its callee, *e.g.* `fs.readFile`. We built a service to gather these names along with their identification. The compiler queries this service to present to the developer an estimated identification. After the compilation, it sends back the identification corrected by the developer to refine the future estimations. In future works, we would like to study the possibility for such a service to assist, and ease the compilation process.

5.3.2 Generation of chains

The compositions of continuations and Dues are arranged differently. Continuations structure the execution flow as a tree, while a chain of Dues imposes to arrange it sequentially. A parent continuation can execute several children, while a Due allow to chain only one. The second compilation step is to identify the imbrications of continuations, and trim the extra branches to transform them into chains.

If a continuation has more than one child, the compiler tries to find a single legitimate child to form the longest chain possible. This legitimate child is the only parent among its siblings. If there are several parents among the children, none are the legitimate child. The non legitimate children start a new tree. This step transform each tree of continuations into several chains of continuations that translate into sequences of Dues. The code generation from these chains is straightforward from the equivalence.

5.4 Evaluation

To validate our compiler, we compile several Javascript projects likely to contain continuations. We present the results of these tests.

The compilation of a project requires user interaction. To conduct the test in a reasonable time, we limit the test set to a minimum. We search the *Node Package Manager* database to restrict the set to *Node.js* projects. We refine the selection to web applications depending on the web framework *express*, but not on the most

[8] `compiler-due.apps.zone52.org`

common Promises libraries such as *Q* and *Async*. We refine further the selection to projects using the test frameworks *mocha* in its default configuration. We use these tests to validate the compiler. The test set contains 64 projects. This subset is very small, and cannot represent the wide possibilities of Javascript. However, we believe it is sufficient to represent a majority of common cases.

For each project, we verify that is is correctly tested, and passes the tests. During the compilation, we identify the compatible continuations among the detected callbacks. We apply the unmodified test on the compilation result. The compilation result should pass the tests as well. This is not a strong validation, but it assures the compiler to work as expected in most common cases.

Of the 64 projects tested, almost a half, does not contain any compatible continuations. We reckon that these projects use continuations the compiler is unable to detect. The other projects were rejected by the compiler because they contain `with` or `eval` statements, they use Promises libraries we didn't filter previously. 9 projects compiled successfully. The compiler did not fail to compile any project of the initial test set.

Over the 9 successfully compiled projects, the compiler detected 172 callbacks. We manually identified 56 of them to be compatible continuations. The false positives are mainly the listeners that the web applications register to react to user requests.

One project contains 20 continuations, the others contains between 1 and 9 continuations each. On the 56 continuations, 36 are single. The others 20 continuations belong to imbrications of 2 to 4 continuations. The result of this evaluation prove the compiler to be able to successfully transform imbrications of continuations.

On the 64 projects composing the test set

29 (45.3%) do not contain any compatible continuations,

10 (15.6%) are not compilable because they contain `with` or `eval` statements,

5 (7.8%) use less common asynchronous libraries we didn't filter previously,

4 (6.3%) are not syntactically correct,

4 (6.3%) fail their tests before the compilation,

3 (4.7%) are not tested, and

10 (14.0%) compile successfully.

The compiler do not fail to compile any project. The details of these projects are available in Appendix **??**.

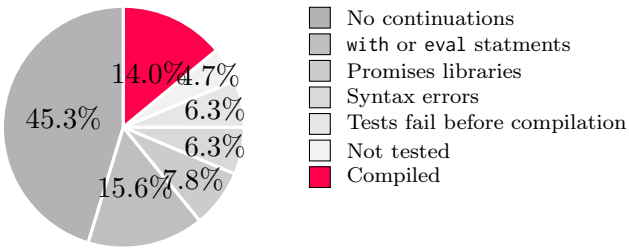


Figure 5.1 – Compilation results distribution

Chapter 6

Pipeline isolation

The previous chapter presented a compiler to identify and extract the underlying pipeline in a Javascript application. However, all the operations are not independent, and cannot be executed in parallel, to support the performance scalability. This chapter present the second contribution of this thesis. The equivalence between a memory shared among all the operations and independent memory for each operation in a pipeline. It tackles the problems arising from the translation of the global memory synchronization into message passing.

This equivalence is implemented as a compiler, improving upon the previous one. The compiler transforms a Javascript application into a network of independent parts communicating by message streams and executed in parallel. We named these parts *fluxions*, by contraction between a flux and a function.

Section 6.1 describes the execution model that executes fluxions in parallel, and assure their communications. The compiler, and the equivalence are described in section 6.2. Section 6.3 a real-case test of compilation, and expose the limits of this compiler.

6.1 Fluxional execution model

In this section, we present an execution model to provide scalability to web applications. To achieve this, the execution model provides a granularity of parallelism at the function level. Functions are encapsulated in autonomous execution containers with their state, so as to be reallocated and executed in parallel. This execution model is close to the actors model, as the execution containers are independent and communicate by messages. The communications are assimilated to stream of messages, similarly to the dataflow programming model. It allows to reason on the

throughput of these streams, and to react to load increases.

The fluxional execution model executes programs written in our high-level fluxionnal language, whose grammar is presented in figure 6.1. An application ⟨program⟩ is partitioned into parts encapsulated in autonomous execution containers named *fluxions* ⟨flx⟩. In the following paragraphs, we present the *fluxions*. Then we present the messaging system to carry the communications between *fluxions*. Finally, we present an example application using this execution model.

6.1.1 Fluxions

A *fluxion* ⟨flx⟩ is named by a unique identifier ⟨id⟩ to receive messages, and might be part of one or more groups indicated by tags ⟨tags⟩. A *fluxion* is composed of a processing function ⟨fn⟩, and a local memory called a *context* ⟨ctx⟩. At a message reception, the *fluxion* modifies its *context*, and sends messages on its output streams ⟨streams⟩ to downstream *fluxions*. The *context* handles the state on which a *fluxion* relies between two message receptions. In addition to message passing, the execution model allows *fluxions* to communicate by sharing state between their *contexts*. The fluxions that need to synchronize together 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 (`>>`) for *start* rupture points and simple arrow (`->`) for *post* rupture points. The two types of rupture points are further detailed in section 6.2.1.1.

6.1.2 Messaging system

The messaging system assures the stream communications between fluxions. It carries messages based on the names of the recipient fluxions. After the execution of a fluxion, it queues the resulting messages for the event loop to process.

The execution cycle of an example fluxional application is illustrated in figure 6.2. Circles represent registered fluxions. The source code for this application is in listing 6.1 and the fluxional code for this application is in listing 6.2. 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,

⟨program⟩ ⊨ ⟨flx⟩ | ⟨flx⟩ *eol* ⟨program⟩
⟨flx⟩ ⊨ `flx` ⟨id⟩ ⟨tags⟩ ⟨ctx⟩ *eol* ⟨streams⟩ *eol* ⟨fn⟩
⟨tags⟩ ⊨ `&` ⟨list⟩ | *empty string*
⟨streams⟩ ⊨ `null` | ⟨stream⟩ | ⟨stream⟩ *eol* ⟨streams⟩
⟨stream⟩ ⊨ ⟨type⟩ ⟨dest⟩ [⟨msg⟩]
⟨dest⟩ ⊨ ⟨list⟩
⟨ctx⟩ ⊨ `{`⟨list⟩`}`
⟨msg⟩ ⊨ `[`⟨list⟩`]`
⟨list⟩ ⊨ ⟨id⟩ | ⟨id⟩ , ⟨list⟩
⟨type⟩ ⊨ `>>` | `->`
⟨id⟩ ⊨ *Identifier*
⟨fn⟩ ⊨ *imperative language and stream syntax*

Figure 6.1 – Syntax of a high-level language to represent a program in the fluxionnal form

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.

6.1.3 Service example

To illustrate the fluxional execution model, and the compiler, we present in listing 6.1 an example of a simple web application. This application reads a file, and sends it back along with a request counter.

```
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 6.1 – Example web application

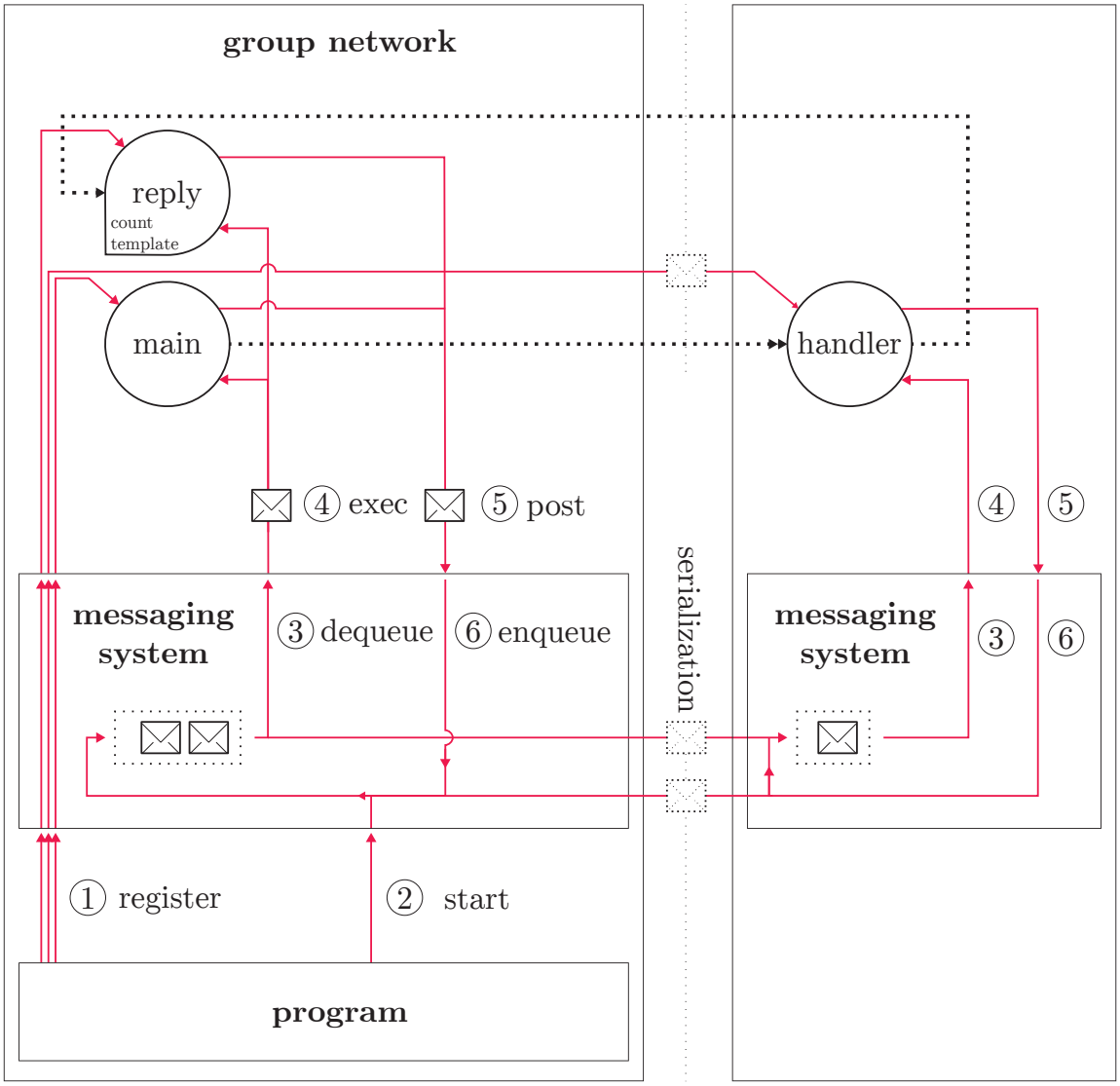


Figure 6.2 – The fluxionnal execution model in details

The `handler` function, line 5 to 11, receives the input stream of request. The `count` variable at line 3 increments the request counter. This object needs to be persisted in the fluxion *context*. The `template` function 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 6.2 which is illustred in Figure 6.2.

```
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 6.2 – 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.

The last fluxion, `reply`, depends on its context to holds the variable `count` and the function `template`. It also depends on the variable `res` created by the first fluxion, `main`. This variable is carried by the stream through the chain of fluxion to the fluxion `reply` that depends on it. This variable holds the references to the network sockets. It is the variable the group `network` depends on.

Moreover, if the last fluxion, `reply`, did not relied on the variable `count`, the group `network` would be stateless. The whole group could be replicated as many time as needed.

This execution model allows to parallelize the execution of an application. Some parts are arranged in pipeline, like the fluxion `handler`, some other parts are replicated, 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.

6.2 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 6.1.

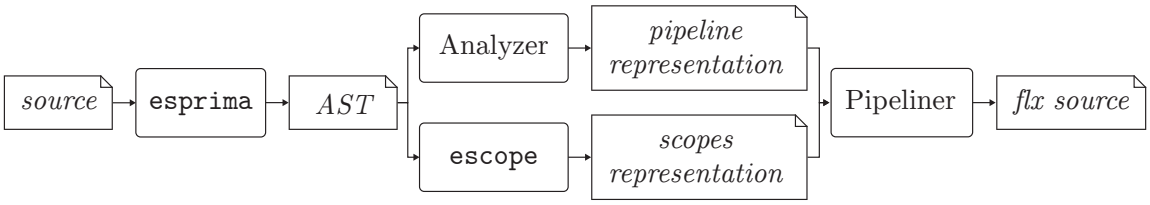


Figure 6.3 – Compilation chain

The chain of compilation is described in figure 6.3. 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 6.2.1 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 6.2.2 explains this second compilation step.

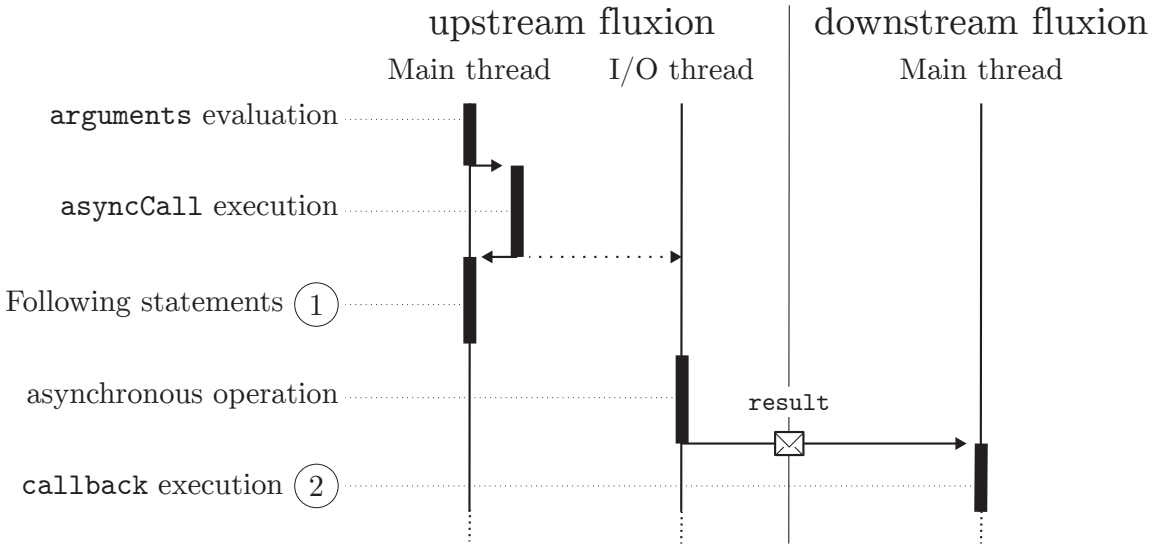
6.2.1 Analyzer step

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.

6.2.1.1 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 6.4 shows an example of a rupture point with the execution of the two application parts isolated into fluxions. The two application parts are the caller of the asynchronous function call on one hand, and the callback provided to the asynchronous function call on the other hand.




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

Figure 6.4 – 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. 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 6.1, 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 6.1, between the call to `fs.readFile()`, and its continuation `reply`.

6.2.1.2 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.

6.2.2 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 6.5.

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.

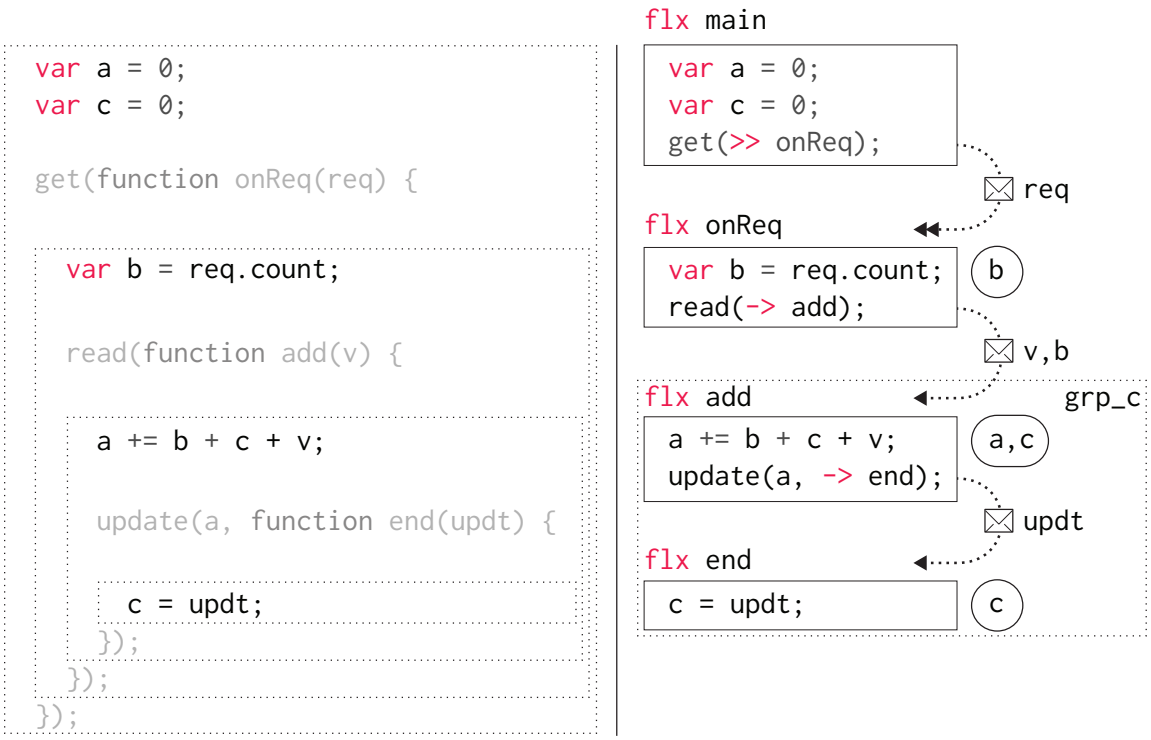


Figure 6.5 – Variable management from Javascript to the high-level fluxionnal language

In figure 6.5, 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 6.5, 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 a same tag. And it adds this variable to the contexts of each fluxions.

In figure 6.5, 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.

6.3 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[1]. 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.

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 6.3 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;
```

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

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

Listing 6.3 – 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`.

6.3.1 Compilation

We compile this application with the compiler detailed in section 6.2. 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 6.4. The compiler detects a rupture point : the function `getRawBody` and its anonymous callback, line 11. It encapsulates this callback in a fluxion named `anonymous_1000`. The callback is replaced with a stream placeholder to send the message stream to this downstream fluxion. The variables `req`, and `next` are 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, the compilation result works as expected. The variables used in the fluxion, `req` and `next`, are still shared between the two fluxions. Our goal is to isolate the two fluxions, to be able to safely parallelize their executions.

```
1  flx main
2  >> anonymous_1000 [req, next]
3    var express = require('express'),
4        app = express(),
5        routes = require('gifsockets-middleware'), //
6        getRawBody = require('raw-body');
7
8    function bodyParser(limit) { //
9      return function saveBody(req, res, next) { //
10       getRawBody(req, { //
11         expected: req.headers['content-length'], //
```

```
12          limit: limit
13        }, >> anonymous_1000);
14      };
15    }
16
17    app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.writeTextToImages); //
18    app.listen(8000);
19
20  flx anonymous_1000
21  -> null
22    function (err, buffer) { //
23      req.body = buffer; //
24      next(); //
25    }
```

Listing 6.4 – Compilation result of gifsockets-server

6.3.2 Isolation

In listing 6.4, 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. This test highlights the current limitations of the compiler, and presents future works to circumvent them.

6.3.2.1 Variable `req`

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

6.3.2.2 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 6.5 – 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 6.5. 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.

6.3.3 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.

Chapter 7

Futur Works

Chapter 8

Conclusion

Appendix A

Language popularity

A.1 PopularitY of Programming Languages (PYPL)

[1] The PYPL index uses Google trends[2] as a leading indicator of the popularity of a programming language. It search for the trend for each programming language by counting the number of searches of this language and the word "tutorial".

PYPL for May 2015

[1] http://pypl.github.io/PYPL.html
[2] https://www.google.com/trends/

Rank	Change	Language	Share	Trend
1		Java	24.1%	-0.9%
2		PHP	11.4%	-1.6%
3		Python	10.9%	+1.3%
4		C#	8.9%	-0.7%
5		C++	8.0%	-0.2%
6		C	7.6%	+0.2%
7		Javascript	7.1%	-0.6%
8		Objective-C	5.7%	-0.2%
9		Matlab	3.1%	+0.1%
10	2× ↑	R	2.8%	+0.7%
11	5× ↑	Swift	2.6%	+2.9%
12	1× ↓	Ruby	2.5%	+0.0%
13	3× ↓	Visual Basic	2.2%	-0.6%
14	1× ↓	VBA	1.5%	-0.1%
15	1× ↓	Perl	1.2%	-0.3%
16	1× ↓	lua	0.5%	-0.1%

A.2 TIOBE

[3]

The TIOBE index uses many search engines as an indicator of the current popularity of programming languages. It count the number of pages each search engine find when queried with the language name and the word "programming". This indicator indicates the number of resources available, and the discussions about a given programming language.

Javascript was the most rising language of 2014 in the TIOBE index.

TIOBE for April 2015

[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Apr 2015	Apr 2014	Change	Programming Language	Ratings	Change
1	2	↑	Java	16.041%	-1.31%
2	1	↓	C	15.745%	-1.89%
3	4	↑	C++	6.962%	+0.83%
4	3	↓	Objective-C	5.890%	-6.99%
5	5		C#	4.947%	+0.13%
6	9	↑	JavaScript	3.297%	+1.55%
7	7		PHP	3.009%	+0.24%
8	8		Python	2.690%	+0.70%
9	-	2× ↑	Visual Basic	2.199%	+2.20%

A.3 Programming Language Popularity Chart

[4]

The programming language popularity chart indicates the activity of a given language in the online communities. It uses two indicators to rank languages : the number of line changed in github of, and the number of questions tagged with a certain language.

Javascript is ranked number one in this index. The Javascript community is particularly active online, and in the open source.

indeed.com

A.4 Black Duck Knowledge

[5]

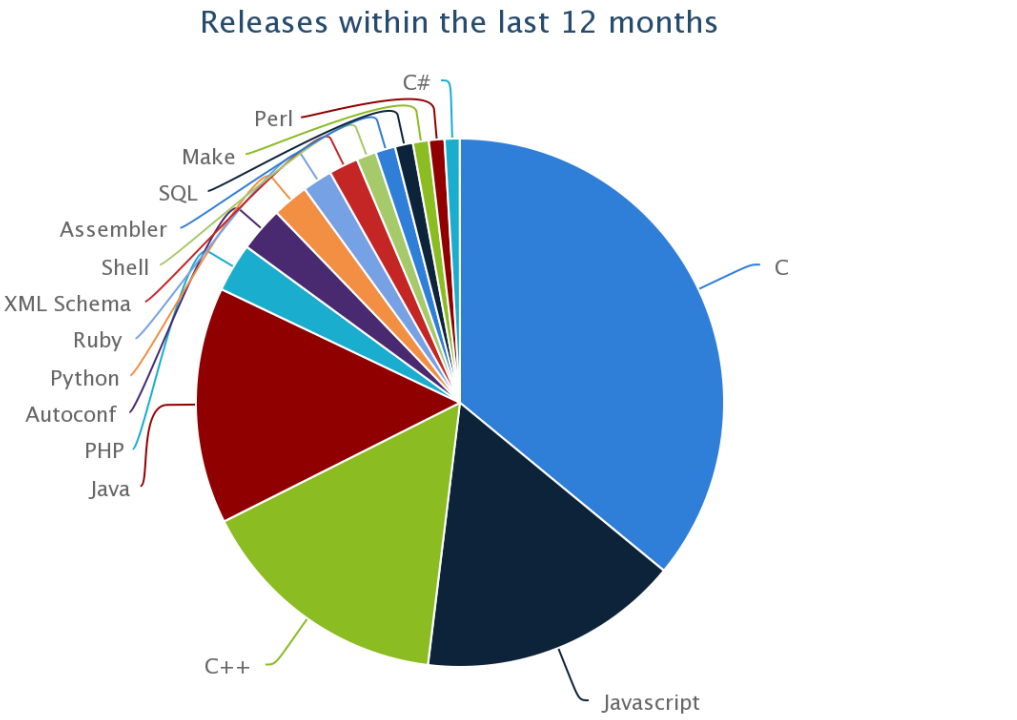
The black-duck, which analyze the usage of language on many forges, and collaborative hosts, rank Javascript number 2, after C, and with about the same usage as C++.

github.com sourceforge.net cpan.org rubyforge7.org planetsourcecode.com ddj.com

[4] http://langpop.corger.nl

[5] https://www.blackducksoftware.com/resources/data/this-years-language-use

Language	%
C	34.80
Javascript	15.45
C++	15.13
Java	14.02
PHP	2.87
Autoconf	2.65
Python	2.15
Ruby	1.77
XML Schema	1.73
Shell	1.18
Assembler	1.16
SQL	1.07
Make	0.94
Perl	0.92
C#	0.90



A.5 Github

http://githut.info/

A.6 HackerNews Poll

https://news.ycombinator.com/item?id=3746692

Language	Count
Python	3335
Ruby	1852
JavaScript	1530
C	1064
C#	907
PHP	719
Java	603
C++	587
Haskell	575
Clojure	480
CoffeeScript	381
Lisp	348
Objective C	341
Perl	341
Scala	255
Scheme	202
Other	195
Erlang	171
Lua	150
Smalltalk	130
Assembly	116
SQL	112
Actionscript	109
OCaml	88
Groovy	83
D	79
Shell	76
ColdFusion	51
Visual Basic	47
Delphi	45
Forth	41
Tcl	34
Ada	29
Pascal	28
Fortran	26
Rexx	13
Cobol	12

Bibliography

[1] D Abadi, D Carney, and U Cetintemel. "Aurora: a data stream management system". In: *Proceedings of the ...* (2003).

[2] DJ Abadi, Y Ahmad, and M Balazinska. "The Design of the Borealis Stream Processing Engine." In: *CIDR* (2005).

[3] DJ Abadi and D Carney. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal— ...* (2003).

[4] A Adya, J Howell, and M Theimer. "Cooperative Task Management Without Manual Stack Management." In: *USENIX Annual Technical Conference* (2002).

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

[6] SP Amarasinghe, JAM Anderson, MS Lam, and CW Tseng. "An Overview of the SUIF Compiler for Scalable Parallel Machines." In: *PPSC* (1995).

[7] GM Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint ...* (1967).

[8] LO Andersen. "Program analysis and specialization for the C programming language". In: (1994).

[9] H Balakrishnan and M Balazinska. "Retrospective on aurora". In: *The VLDB Journal* (2004).

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

[11] JR von Behren, J Condit, and EA Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers)." In: *HotOS* (2003).

[12] R Von Behren, J Condit, and F Zhou. "Capriccio: scalable threads for internet services". In: *ACM SIGOPS ...* (2003).

[13] M Bodin and A Charguéraud. “A trusted mechanised JavaSript specification”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014).

[14] I Buck, T Foley, and D Horn. “Brook for GPUs: stream computing on graphics hardware”. In: *... on Graphics (TOG)* (2004).

[15] S Chandrasekaran and O Cooper. “TelegraphCQ: continuous dataflow processing”. In: *Proceedings of the ...* (2003).

[16] J Chen, DJ DeWitt, F Tian, and Y Wang. “NiagaraCQ: A scalable continuous query system for internet databases”. In: *ACM SIGMOD Record* (2000).

[17] 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.

[18] C Consel, H Hamdi, and L Réveillère. “Spidle: a DSL approach to specifying streaming applications”. In: *Generative ...* (2003).

[19] Melvin E. Conway. “Design of a separable transition-diagram compiler”. In: *Communications of the ACM* 6.7 (July 1963), pp. 396–408. DOI: 10.1145/366663.366704.

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

[21] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975.

[22] Edsger W. Dijkstra. “Letters to the editor: go to statement considered harmful”. In: *Communications of the ACM* 11.3 (Mar. 1968), pp. 147–148. DOI: 10.1145/362929.362947.

[23] Edsger W. Dijkstra. “The structure of the “THE”-multiprogramming system”. In: *Communications of the ACM* 11.5 (May 1968), pp. 341–346. DOI: 10.1145/363095.363143.

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

[25] JI Fernández-Villamor. “Microservices-Lightweight Service Descriptions for REST Architectural Style.” In: *... 2010-Proceedings of ...* (2010).

[26] D Flanagan. *JavaScript: the definitive guide*. 2006.

[27] P Gardner and G Smith. "JuS: Squeezing the sense out of javascript programs". In: *JSTools@ ECOOP* (2013).

[28] PA Gardner, S Maffeis, and GD Smith. "Towards a program logic for JavaScript". In: *ACM SIGPLAN Notices* (2012).

[29] JJ Garrett. "Ajax: A new approach to web applications". In: (2005).

[30] SD Gribble, M Welsh, and R Von Behren. "The Ninja architecture for robust Internet-scale systems and services". In: *Computer Networks* (2001).

[31] NJ Gunther. "A New Interpretation of Amdahl's Law and Geometric Scalability". In: *arXiv preprint cs/0210017* (2002).

[32] NJ Gunther. "A simple capacity model of massively parallel transaction systems". In: *CMG-CONFERENCE-* (1993).

[33] NJ Gunther. "Understanding the MP effect: Multiprocessing in pictures". In: *In other words* (1996).

[34] JL Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* (1988).

[35] B Hackett and S Guo. "Fast and precise hybrid type inference for JavaScript". In: *ACM SIGPLAN Notices* (2012).

[36] CT Haynes, DP Friedman, and M Wand. "Continuations and coroutines". In: *... of the 1984 ACM Symposium on ...* (1984).

[37] B He, M Yang, Z Guo, R Chen, and B Su. "Comet: batched stream processing for data intensive distributed computing". In: *... on Cloud computing* (2010).

[38] C Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial intelligence* (1977).

[39] 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).

[40] C. A. R. Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. DOI: 10.1145/359576.359585.

[41] C. A. R. Hoare. "Monitors: an operating system structuring concept". In: *Communications of the ACM* 17.10 (Oct. 1974), pp. 549–557. DOI: 10.1145/355620.361161.

[42] YW Huang, F Yu, C Hang, and CH Tsai. "Securing web application code by static analysis and runtime protection". In: *Proceedings of the 13th ...* (2004).

[43] M Isard, M Budiu, Y Yu, A Birrell, and D Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS Operating ...* (2007).

[44] D Jang and KM Choe. "Points-to analysis for JavaScript". In: *Proceedings of the 2009 ACM symposium on Applied ...* (2009).

[45] N Jovanovic, C Kruegel, and E Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities". In: *Security and Privacy, 2006 ...* (2006).

[46] Gilles Kahn. "The semantics of a simple language for parallel programming". In: (1974).

[47] Gilles Kahn and David Macqueen. *Coroutines and Networks of Parallel Processes*. en. Tech. rep. 1976, p. 20.

[48] S Krishnamurthy and S Chandrasekaran. "TelegraphCQ: An architectural status report". In: *IEEE Data Eng. ...* (2003).

[49] MN Krohn, E Kohler, and MF Kaashoek. "Events Can Make Sense." In: *USENIX Annual Technical Conference* (2007).

[50] L Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* (1978).

[51] D Logothetis, C Olston, and B Reed. "Stateful bulk processing for incremental analytics". In: *Proceedings of the 1st ...* (2010).

[52] S Maffeis, JC Mitchell, and A Taly. "An operational semantics for JavaScript". In: *Programming languages and systems* (2008).

[53] S Maffeis, JC Mitchell, and A Taly. "Isolating JavaScript with filters, rewriting, and wrappers". In: *Computer Security–ESORICS 2009* (2009).

[54] WR Mark and RS Glanville. "Cg: A system for programming graphics hardware in a C-like language". In: *... Transactions on Graphics (...* (2003).

[55] ND Matsakis. "Parallel closures: a new twist on an old idea". In: *Proceedings of the 4th USENIX conference on Hot ...* (2012).

[56] MD McCool. "Structured parallel programming with deterministic patterns". In: *Proceedings of the 2nd USENIX conference on Hot ...* (2010).

[57] M Migliavacca and D Eyers. "SEEP: scalable and elastic event processing". In: *Middleware'10 Posters ...* (2010).

[58] G Moore. “Cramming More Components Onto Integrated Circuits”. In: *Electronics* 38 (1965), p. 8.

[59] JP Morrison. *Flow-Based Programming*. 1994, pp. 1–377.

[60] JF Naughton, DJ DeWitt, and D Maier. “The Niagara internet query system”. In: *IEEE Data Eng. ...* (2001).

[61] R Nelson. “Including queueing effects in Amdahl’s law”. In: *Communications of the ACM* (1996).

[62] L Neumeyer and B Robbins. “S4: Distributed stream computing platform”. In: *Data Mining Workshops ...* (2010).

[63] VS Pai, P Druschel, and W Zwaenepoel. “Flash: An efficient and portable Web server.” In: *USENIX Annual Technical Conference* (1999).

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

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

[66] Z Qian, Y He, C Su, Z Wu, and H Zhu. “Timestream: Reliable stream computation in the cloud”. In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys ’13)* (2013).

[67] C Radoi, SJ Fink, R Rabbah, and M Sridharan. “Translating imperative code to MapReduce”. In: *Proceedings of the 2014 ...* (2014).

[68] DP Reed. “"" Simultaneous"" Considered Harmful: Modular Parallelism.” In: *HotPar* (2012).

[69] MC Rinard and PC Diniz. “Commutativity analysis: A new analysis framework for parallelizing compilers”. In: *ACM SIGPLAN Notices* (1996).

[70] Tiago Salmito, Ana Lucia de Moura, and Noemi Rodriguez. “A Flexible Approach to Staged Events”. English. In: *2013 42nd International Conference on Parallel Processing*. IEEE, Oct. 2013, pp. 661–670. DOI: 10.1109/ICPP.2013.80.

[71] Tiago Salmito, Ana Lúcia de Moura, and Noemi Rodriguez. “A stepwise approach to developing staged applications”. In: *The Journal of Supercomputing* (Jan. 2014). DOI: 10.1007/s11227-014-1110-4.

[72] GD Smith. “Local reasoning about web programs”. In: (2011).

[73] M Sridharan, J Dolby, and S Chandra. “Correlation tracking for points-to analysis of JavaScript”. In: *ECOOP 2012–Object- ...* (2012).

[74] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". English. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. DOI: 10.1147/sj.132.0115.

[75] GJ Sussman and GL Steele Jr. "Scheme: A interpreter for extended lambda calculus". In: *Higher-Order and Symbolic Computation* (1998).

[76] W Thies, M Karczmarek, and S Amarasinghe. "StreamIt: A language for streaming applications". In: *Compiler Construction* (2002).

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

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

[79] M Welsh, D Culler, and E Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review* (2001).

[80] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. "Design Rule Hierarchies and Parallelism in Software Development Tasks". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Nov. 2009, pp. 197–208. DOI: 10.1109/ASE.2009.53.

[81] D Yu, A Chander, N Islam, and I Serikov. "JavaScript instrumentation for browser security". In: *ACM SIGPLAN Notices* (2007).

[82] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey, Frank McSherry, Kannan Achan, and Christophe Poulain. "Some sample programs written in DryadLINQ". In: *Microsoft Research* (2009).