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

Etienne Brodu

December 29, 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

List of Figures

List of Tables

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 6 concludes this thesis, and draws the possible perspectives beyond this work.

Chapter 2

Context and objectives

Contents

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.

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.

2.1 The Web as a Platform

Similarly to operating systems, Web browsers started as software products with extension capabilities that transformed it into a platform. 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 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 in 1991, Ruby in 1993, Java in 1994, PHP and Javascript in 1995.

With the democratisation of programming, the involvement of the community became critical for the adoption, evolution and maturation of a language. Java thrived in the software industry, but lose the hype that drove the community innovation and creativity. Now, it 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. Wordpress, a PHP publishing platform, is an economical success.

Since a few years, Javascript is slowly becoming the main language for web development. The next paragraph present its evolution in the industry and the community.

2.1.1.1 The Ugly Duckling

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

— B. Stroustrup[1]

Javascript was released as a scripting engine in Netscape Navigator around September 1995 and later in its concurrent, Internet Explorer. The competition between the two was fragmenting the Web. Web pages 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 – the first standard for Javascript in June 1997.

illustration: the ugly duckling

The initial release of Javascript was designed by Brendan Eich 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?[2] Is Javascript here to stay?[3] Why Javascript Is Doomed.[4] Why JavaScript Makes Bad Developers.[5] JavaScript: The World's Most Misunderstood Programming

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

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

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

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

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

Language[6] Why Javascript Still Sucks[7] 10 things we hate about JavaScript[8] Why do so many people seem to hate Javascript?[9]

But this situation evolved drastically since. All web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime [43]. This position became an incentive to make it fast (V8, ASM.js) and convenient (ES6, ES7). Any Javascript code is open, allowing the community to pick, improve and reproduce the best techniques [10]. Javascript is distributed freely, with all the tools needed to reproduce and experiment on the largest communication network in history. 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. All these reasons made the popularity of the Web and Javascript.

2.1.1.2 The Rise of 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[11]

Javascript was initially used for short interactions on web pages. Nowadays, there is a lot of web-based applications replacing desktop applications, like mail client, word processor, music player, graphics editor...

ECMA International allowd this progression by releasing several versions to give Javascript a more complete and solid base as a programming language. Moreover, Asynchronous Javascript And XML (Ajax) allows to dynamically reload the content inside a web page, hence improving the user experience [51]. It allows Javascript to develop richer applications inside the browser, from user interactions to network communications. The community released frameworks to assist the development of these larger applications. Prototype[12] and DOJO[13] are early famous examples, and

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

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

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

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

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

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

[12] http://prototypejs.org/

[13] https://dojotoolkit.org/

illustration: Javascript with superpowers

later jQuery[14] and underscore[15].

Since 2004, the Web Hypertext Application Technology Working Group[16] worked 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 simultaneous release of HTML5, ECMAScript 5 and V8, around 2009, represent a milestone in the development of web-based applications. Javascript became the *de facto* programming language to develop on this rising application platform that is the Web[17].

Javascript is now 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 the most important language in the times to come. Especially that Javascript now allows to build the server side of web applications as well. The next section presents the realities and technical challenges to assure the performance of web services against billions of users.

*

⚠

Frise chronologique Javascript

2.1.2 Highly Concurrent Web Servers

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

The Internet allows communication at an unprecedented scale. There is more than 16 billions connected devices, and it is growing fast[18] [73]. A large web application like google search receives about 40 000 requests per seconds[19]. 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[20]. In the 2010s, the limit is set at 10 millions simultaneous connections[21]. With the growing number of connected devices on the internet, concurrency is a very important property in the design of web applications.

[14] https://jquery.com/
[15] http://underscorejs.org/
[16] https://whatwg.org/
[17] http://blog.codinghorror.com/javascript-the-lingua-franca-of-the-web/
[18] http://blogs.cisco.com/news/cisco-connections-counter
[19] http://www.internetlivestats.com/google-search-statistics/
[20] http://www.kegel.com/c10k.html
[21] http://c10m.robertgraham.com/p/manifesto.html

2.1.2.1 Event-Loop Execution Model

Javascript is often associated with an event-based paradigm to react to concurrent user interactions. This event-based paradigm proved to be very efficient as well for a web service to react to concurrent requests. In 2009, Joyent released Node.js to build real-time web services with this paradigm.

*

⚠ TODO schema of an event-loop

At reception, each request from a client queues an event waiting to be processed. The event-loop unqueues these events one at a time. Processing an event can imply to query remote resources, which respond asynchronously additional event to queue. Or an event can respond directly to the client, ending this chain of asynchronous events.

This execution model allows a high concurrency to respond to a high number of users simultaneously. This concurrency needs to be scalable to adapt to the 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 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.

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.

The Javascript event-loop requires a global memory to assure the interdependency of the tasks. This thesis argues that there exists an equivalence between the event-loop model and the pipeline execution model.

2.1.2.2 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.

*

Each stage is organized in a similar fashion than the event-loop presented in section 2.1.2.1. 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.

⚠ TODO schema of a pipeline

* 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 world wide 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[22]. 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.

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[23]. The development team opt for a popular and accessible language to leverage the advantage of its community. It is only after a

[22] https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/
[23] https://www.cbinsights.com/blog/startup-failure-post-mortem/

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 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 [124]

With the growth of Software as a Service (SaaS) on the web, the same company carries both development and exploitation of an application at scale of unprecedented size. It revealed the importance of previously unknown economic constraints. To assure the continuous growth and sustainability of an application, it needs to address two contradictory goals : development productivity and performance efficiency. These goals needs to be enforced by the platform supporting the application to build good development habits for the developers. A platform designates any solution that allows to build an application on top of it, including programming languages, compilers, interpreters, frameworks, runtime libraries and so on.

75% of your budget is dedicated to software maintenance.[1] The productivity of a platform is the degree to which developers can quickly produce new and modify existing software. It impacts the maintainability of the applications and relies on the modularity enforced by its platform. Especially, higher order programming is crucial to build and compose modules productively. It relies either on mutable states, or immutable states, but hardly on a combination of both.

However, neither mutable nor immutable states allows performance efficiency. Mutable states leads to synchronization overhead at a coarser-grain level, while immutable states leads to communication overhead at a finer-grain level. Efficiency relies on a combination of synchronization at a fine-grain level, and immutable message passing at a coarse-grain level. This combination breaks the modularity, hence the productivity of an application. A company has no choice but to commit huge development efforts to get efficient performances.

illustration: virtuous circle between community and industry

Moreover, a balance between productivity and efficiency is required for a platform to enter a virtuous circle of adoption. The productivity is required to be appealing to gather a community to support the ecosystem around the platform. This community is appealing for the industry as a hiring pool. Additionally, the efficiency is required to be adopted by the industry to be economically viable. And the industrial relevance provides the reason for this ecosystem to exist and the community to gather.

This chapter presents a broad view of the state of the art in the compromises between productivity and efficiency. It defines software productivity, efficiency, and adoption in section 3.1 and all the underlying concepts, such as higher order programming and state mutability. It then analyzes different platforms according to their focus. platforms focusing on productivity are addressed in section 3.2, those

[1] `http://www.castsoftware.com/glossary/software-maintainability`

focusing on efficiency in section 3.3 and those focusing on a compromise between the two in section 3.4.

3.1 Definitions

The continuous growth and sustainability of a platform relies on three criteria. This section defines these tree criteria, as well as all the underlying concepts.

- Productivity
- Efficiency
- Adoption

3.1.1 Productivity

The productivity of a platform is the degree to which developers can quickly produce new and modify existing software. For a platform to be productive, it needs to enforce modularity directly in the design of applications. Productivity later leads to maintainability.

3.1.1.1 Modularity

Modularity is about encapsulating subproblems and composing them to allow greater design to emerge. It allows to limit the understanding required to contribute to a module [121], which helps developers to repair and enhance the application. Additionally, it reduces development time by allowing several developers to simultaneously implement different modules [140, 19].

The criteria to define modules to improve productivity are high cohesion and low coupling [121]. Cohesion defines how strongly the features inside a module are related. Coupling defines the strength of the interdependences between modules. The encapsulation of modules helps increase their cohesion, and their composition helps decrease coupling. Encapsulation and composition improve productivity.

- Encapsulation $\rightarrow$ High Cohesion
- Composition $\rightarrow$ Low Coupling

3.1.1.2 Encapsulation

illustration: spaghetti programming

Boundary Definition Modular Programming stands upon Structured Programming [35]. It draws clear interfaces around a piece of implementation so that the execution remains enclosed inside. At a fine level, it helps avoid spaghetti code [38], and at a coarser level, it structures the implementation [39] into modules, or layers.

illustration: lasagna programming

Data Protection Modular programming encapsulates a specific design choice in each module, so that it is responsible for one and only one concern. It isolates its evolution from impacting the rest of the implementation [109, 128, 79]. 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.

3.1.1.3 Composition

Higher-Order Programming Higher-order programming introduces lambda expressions, functions manipulable like any other primary value. They can be stored in variables, or be passed as arguments. It replaces the need for most modern object oriented programming design patterns [2] with Inversion of Control [83], the Hollywood Principle [127], and Monads [136]. Higher-order programming help loosen coupling, thus improve productivity [65].

Closures In languages allowing mutable state, lambda expressions are implemented as closure, to preserve the lexical scope [126]. 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.

Lazy Evaluation Lazy evaluation allows to defer the execution of an expression when its result is needed. The lazy evaluation of a list is equivalent to a stream with a null-sized buffer [134]. It is a powerful tool for structuring modular programs, as the execution is organized as a concurrent pipeline [1]. The stages process independently each element of the stream. But this concurrency requires the isolation of side-effects to avoid conflicts between stages executions.

The criteria to analyze the productivity of platforms are the following.

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

- Encapsulation → High Cohesion

  Boundary definition

  Data protection

- Composition → Low Coupling

  Higher-order programming, Lambda Expressions

  Lazy evaluation, Stream composition

3.1.2 Efficiency

The efficiency of a software project is the relation between the usage made of available resources and the delivered performance. For an application to perform efficiently, the platform based on needs to enforce scalability directly in its design.

Scalability relies on the parallelism allowed by the commutativity of operations execution [26]. An operation is a sequence of statements. Operations are commutative if the order of their executions is irrelevant for the correctness of their results. Commutativity assures the independence of operations.

3.1.2.1 Independence

illustration: Synchronization vs Message-passing

The independence, and commutativity of an operations depends on its accesses to shared state. If the operations doesn't rely on any shared state, it is independent. The independence of operations allows to execute them in parallel, hence to increase performance proportionally to occupied resources [6, 58]. But if they rely on shared state, they need to coordinate the causal scheduling and atomicity of their executions to avoid conflicting accesses. This scheduling between the operations can be defined in two ways.

Synchronization Operations are scheduled sequentially to have the exclusivity on a shared state, or

Message-passing Operations communicate their local modifications of the state to other operations, in a decentralized fashion.

3.1.2.2 Atomicity

An operation is atomic if it happens in a single bulk. The beginning and end are indistinguishable for an external observer. It assures the developer of the invariance of the memory during the operation. It relies either on the causal scheduling of

operations – synchronization – or exclusivity of their memory accesses – message-passing.

3.1.2.3 Granularity

If the operations access the state too frequently, the communication overhead of message passing exceeds the performance gains of parallelism. And if operations access the state too rarely, the synchronization required for sharing state limits the possible parallelism. These two extremes are inefficient. Operations tend to share state closely at a fine-grain level and less at a coarser-grain level. Therefore, efficiency requires the combination of fine-level state sharing to avoid communication overhead, and coarse-level independence to allow parallelization [60, 59, 104, 57]. The threshold determining frequent or rare access to the state determines the granularity level between synchronization and parallelization of tasks.

The criteria to analyze the performance of platforms are the following.

- Fine-level state sharing

  State mutability → Synchronization

- Coarse-level independence

  State immutability → Message-passing

3.1.3 Adoption

An application is sustainable only if the platform used to build it generates reinforcing interactions between a community of passionate and the industry. A platform needs to present a balance between productivity and efficiency to be adopted by both the community and the industry. The productivity is required for a platform to be appealing to gather a community to support the ecosystem around it. And the efficiency is required to be economically viable and needed by the industry, and to provide the reason for this ecosystem to exist. Additionally, the web acts as a tremendous catalyst fueling these interactions.

The criteria to analyze the adoption of platforms are the following.

- Community Support
- Industrial Need

Adoption requires a balance between efficiency and productivity. This incentive to balance between productivity and efficiency is illustrated in figure 3.1. This figure is used throughout this chapter to graphically represent all the platforms analyzed.

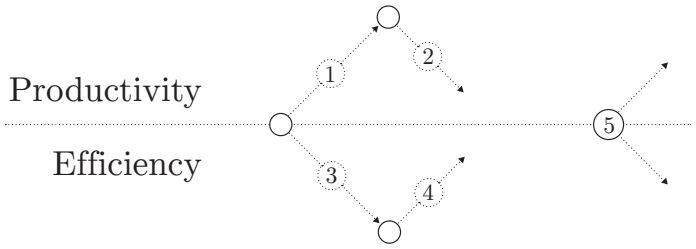


Figure 3.1 – Balance between Efficiency and Productivity

3.2 Productivity Focused Platforms

"It is becoming increasingly important to the data-processing industry to be able to produce [programming systems] at a faster rate, and in a way that modifications can be accomplished easily and quickly."

— W. Stevens, G. Myers, L. Constantine [121].

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 becomes more and more difficult to grasp. Therefore, it is crucial to decompose the system into smaller subsystem easier to grasp individually.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

— Bill Gates

Section 3.2.1 presents the modular programming paradigms, and their programming models, oriented toward productivity. Section 3.2.2 presents the adoption of the implementations of modular programming languages. Section 3.2.3 presents the consequences of the modularity on performance. Finally, section 3.2.4 summarizes the three previous sections in a table.

3.2.1 Modular Programming

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

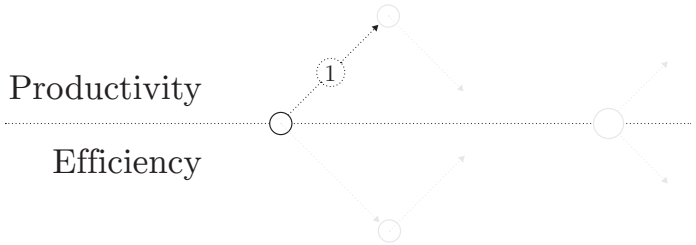


Figure 3.2 – Focus on Productivity

3.2.1.1 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. The implementations of Imperative Programming

3.2.1.2 Object Oriented Programming

illustration: multiple cells communicating

The very first Object-Oriented Programming (OOP) language was Smalltalk [52]. It defined the core concepts as message passing and encapsulation [3]. Nowadays, the emblematic figures in the software industry are C++ C++ [123] and Java [54]. They provide encapsulation with Classes, and allows passing mutable structures for performance reasons. They recently introduced higher-order programming with lambda expressions.

3.2.1.3 Functional Programming

The definition of pure Functional Programming resides in manipulating only expressions 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 [115], Miranda [131], Haskell [77] and Standard ML [100]. They provide encapsulation, higher-order programming and lazy evaluation.

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

3.2.1.4 Multi-Paradigm

The functional programming concepts are also implemented in other languages along with mutable states and object-oriented concepts. Major recent programming languages, including Java 8 and C++ 11, now commonly present **higher-order functions** and **lazy evaluation**. *In fine*, it helps developers to write applications that are more maintainable, and favorable to evolution [78, 132]. These recent multi-paradigms languages such as Javacript, Python and Ruby combine the different paradigms to help developer building applications faster.

Table 3.1 presents a summary of the analysis of the programming models presented in the previous paragraphs.

Model	Composition	Encapsulation	→	Productivity
Imperative Programming	3	4		3
Object-Oriented Programming	5	5		5
Functional Programming	5	5		5
Multi Paradigm	5	5		5

Table 3.1 – Productivity of Modular Programming Platforms

3.2.2 Adoption

The next paragraphs presents the adoption of Javascript, and the other implementations of the presented programming model.

3.2.2.1 Community

Available Resources As of December 2015, Javascript ranks 8th according to the TIOBE Programming Community index, and was the most rising language in 2014. This index measure the popularity of a programming language with the number of

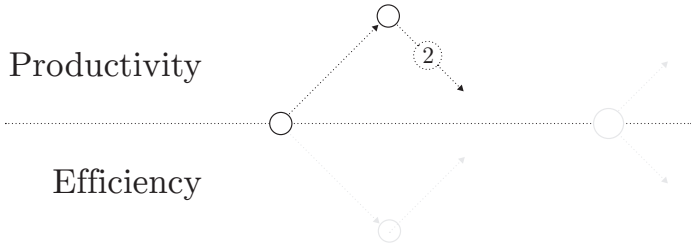


Figure 3.3 – Steering back toward Performence Efficiency

results on many search engines. And it ranks 7th on the PYPL. The PYPL index is based on Google trends to measure the number of requests on a programming language.

From these indexes, the major programming languages are Java, C++, C, C# and Python. These languages are still widely used by their communities and in the industry.

*

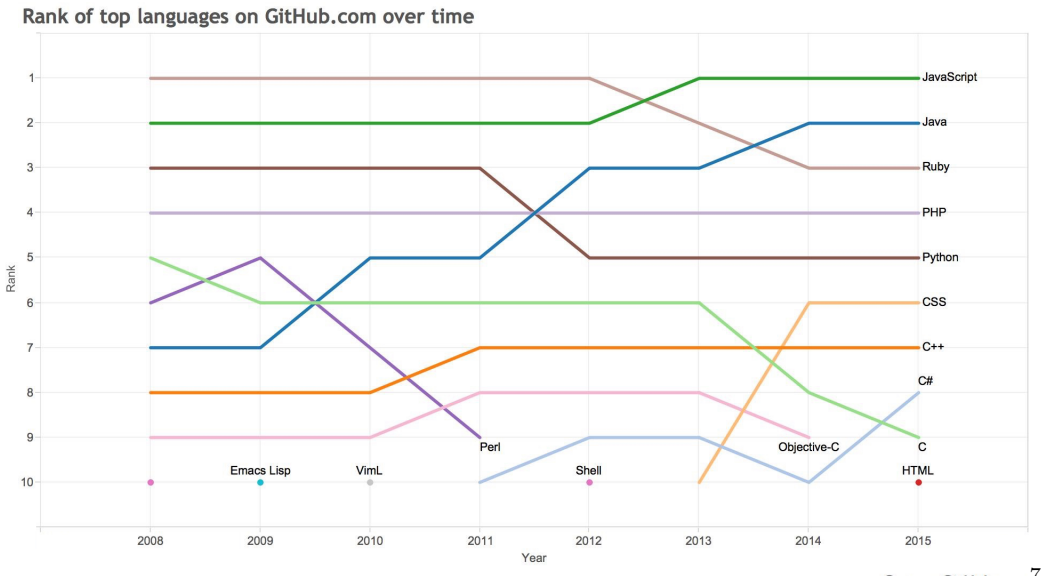
TODO graphical ranking of TIOBE and PYPL

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*[4] and the most cited language on *StackOverflow*[5]. It represents more than 320,000 repositories on *Github*. The second language is Java with more than 220,000 repositories. It is cited in more than 960,000 questions on *StackOverflow* while the second is Java with around 940,000 questions. And according to a survey by *StackOverflow*, it is currently the language the most popular[6]. Moreover, the Javascript package manager, *npm*, has the most important and impressive package repository growth.

*

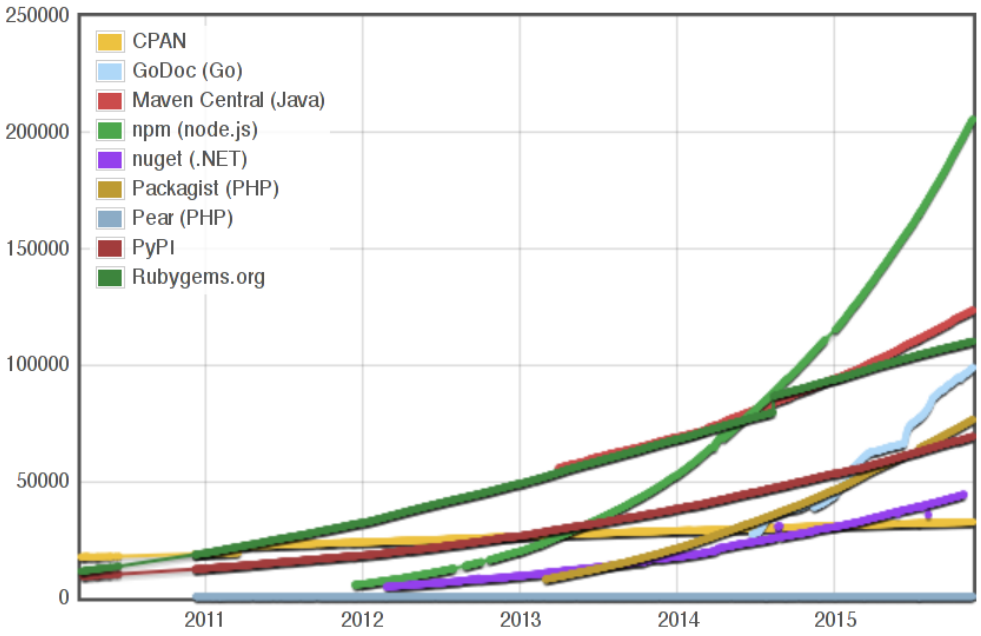
TODO include so survey graph

[4] themostimportantcollaborativedevelopmentplatformgatheringabout9millionsusers.
[5] themostimportantQ&Aplatformfordevelopers.
[6] http://stackoverflow.com/research/developer-survey-2015



[7]

TODO redo this graph, it is ugly.



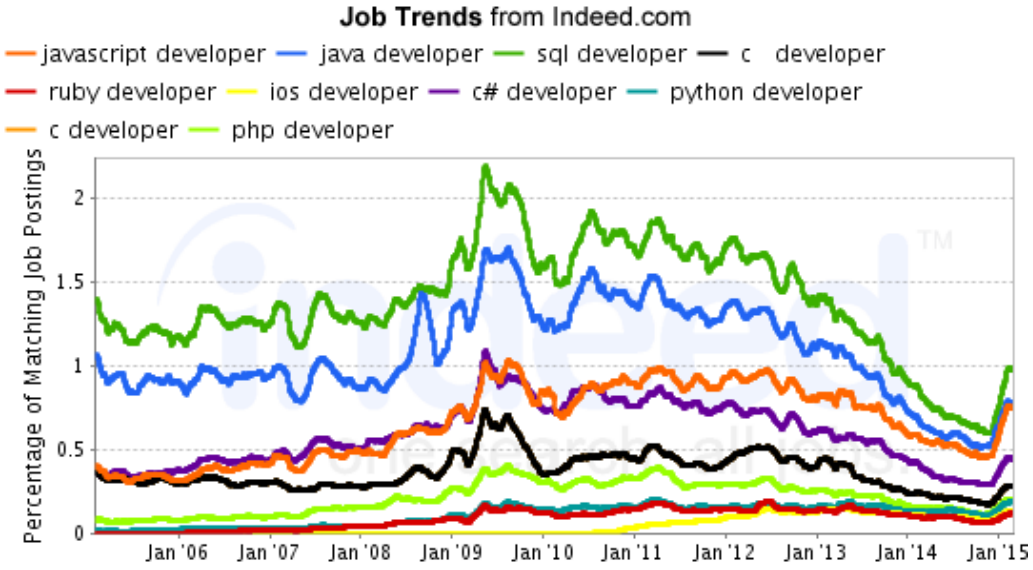
TODO graphical ranking of the tags in StackOverflow

TODO redo this graph, it is ugly.

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

3.2.2.2 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*[8], right after SQL and Java.[9] Moreover, according to *breaz.io*[10], Javascript developers get more opportunities than any other developers. Javascript is increasingly adopted in the software industry.



TODO redo this graph, it is ugly.

Table 3.2 presents a summary of the analysis of the programming models presented in the previous paragraphs.

[8] http://www.indeed.com

[9] 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=

[10] https://breaz.io/

Model	Community support	Industrial need	→	Adoption
Imperative Programming	3	4		3
Object-Oriented Programming	4	4		4
Functional Programming	0	1		1
Multi Paradigm	5	4		4

Table 3.2 – Adoption of Modular Programming Platforms

3.2.3 Efficiency Limitations

Eventually, the presented languages are hitting a wall on their way to performance.

All the languages presented previously provide either mutable state or immutable state on which to rely to assure encapsulation and composition. Functional programming relies on immutable message-passing. It might impacts performance at a fine-grain level because of heavy memory usage. On the other hand, the synchronization required by mutable state is often hard to develop with [3], or avoid parallelism [108, 88].

The only solution to provide performance efficiency is to combine mutable state at a fine-grain level, with synchronization, and immutable state at a coarse-grain level, with message-passing.

The table 3.3 presents the performance limitations of the languages presented in this section. The platforms extending these languages with concurrent or parallel features to provide performances are addressed in the next section.

Model	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Imperative Programming	0	0		0
Object-Oriented Programming	0	0		0
Functional Programming	0	0		0
Multi Paradigm	0	0		0

Table 3.3 – Efficiency of Modular Programming Platforms

3.2.4 Summary

Table 3.4 summarizes the characteristics of the solutions presented in this section.

Model	Productivity	Adoption	Efficiency
Imperative Programming	3	3	0
Object-Oriented Programming	5	4	0
Functional Programming	5	1	0
Multi Paradigm	5	4	0

Table 3.4 – Summary of Modular Programming Platforms

3.3 Efficiency Focused Platforms

Both the academia and the industry proposed solutions with efficiency in mind to cope with the limitations the previous section concludes on. Section 3.3.1 presents the concurrent and parallel programming paradigms, and their programming models. Section 3.3.2 presents the adoption steered by the efficiency of parallel programming. Section 3.3.3 presents the consequences of parallelism on productivity. Finally, section 3.3.4 summarizes the three previous sections in a table.

3.3.1 Concurrency

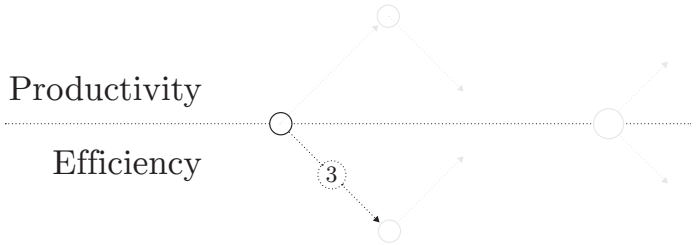


Figure 3.4 – Focus on Efficiency

Web servers need to be able to process huge amount of concurrent operations in a scalable fashion. Concurrency is the ability to make progress on several operations roughly simultaneously. It implies to draw memory boundaries to define independent regions, or to define causality in the execution of tasks. When both boundaries and causality are clearly defined, the tasks are independent and can be scheduled in parallel to make progress strictly simultaneously.

The definition of independent tasks allows the fine level synchronization within a task, and coarse level message passing between the tasks required for performance efficiency. The synchronization of execution at a fine level assures the invariance on the shared state, and avoid communication overhead. The message-passing at a coarser level assures the parallelism. The two are indispensable for efficiency.

3.3.1.1 Concurrent Programming

illustration: feu rouge et rond point

Concurrent programming provides the mechanisms to assure atomicity of concurrent operations. They define the causal scheduling of execution and assure the invariance

of the global memory. There are two scheduling strategies to execute concurrent tasks 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.

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 the unexpected preemption breaks atomicity, the developer needs to lock the shared state to assure atomicity and exclusivity.

The next paragraphs presents the programming model for these scheduling strategy, the event-driven programing model based on cooperative scheduling, and the multi-threading programming model based on preemptive scheduling. Additionally, they present two alternatives to these two main programming models, lock-free data-structures and Fibers.

Event-Driven Programming Event-driven execution model queues concurrent tasks needing access to shared resources. The tasks are explicitely defined by the developer. The concurrent tasks are schedule sequentially to assure exclusivity, and cooperatively to assure atomicity. It is very efficient for highly concurrent applications, as it avoids contention due to waiting for shared resources like disks, or network. Several execution model rely on this execution model, like TAME [88], Node.js[11] and Vert.X[12]. As well as some web servers like Flash [108], Ninja [55] thttpd[13] and Nginx[14].

But the event-driven model is limited in performance. The concurrent tasks share the same memory, and cannot be scheduled in parallel. The next paragraph presents work intending to improve performance by reducing the atomic portions of operations to a minimum.

Lock-Free Data-Structures The wait-free and lock-free data-structures use atomic operations small enough so that locking is unnecessary [90, 69, 67, 68, 8]. They are

[11] `https://nodejs.org/en/`
[12] `http://vertx.io/`
[13] `http://acme.com/software/thttpd/`
[14] `https://www.nginx.com/`

based on instructions provided by transactional memories [63] that combine read and write instructions, They provide concurrent implementations of basic data-structures such as linked list [133, 130], queue [125, 139], tree [112] and stack [66].

However these atomic operations are scheduled sequentially, which limits parallelism. The next paragraphs present multi-threading, which, contrary to the event-driven model, requires the developer to explicitly define atomicity.

Multi-Threading Programming Threads are the small execution containers sharing the same memory execution context within an isolated tasks [39], and scheduled in parallel with fork/join instructions [113, 48, 92]. They execute statements sequentially waiting for completion, and are scheduled preemptively to avoid blocking the global progression. The preemption breaks the atomicity of the execution, and the parallel execution breaks the exclusivity of memory accesses. To restore atomicity and exclusivity, hence assure the invariance, multi-threading programming models provide synchronization mechanisms, such as semaphores [36], guarded commands [37], guarded region [62] and monitors [75].

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 [3].

Cooperative Threads Cooperative threads, or fibers join the advantage of sequential waiting, with the advantage of cooperative scheduling [3, 15]. It avoids splitting the execution into atomic tasks nor use synchronization mechanisms to assure exclusivity. A fiber yields the execution to another fiber to avoid blocking the execution during a long-waiting operation, and recovers it at the same point when the operation finishes. However, developers need to be aware of these yielding operation to preserve the atomicity[15].

Limitation of Concurrent Programming Concurrent programming provides the synchronization required to assure sequentiality of execution within a task and the causal ordering between tasks. However, multi-threading imposes sequentiality between tasks as well. This global sequentiality is excessive ; it impacts performance, and is difficult to manage efficiently.

The causal ordering between tasks proposed by the event-driven execution model is sufficient to assure correctness of execution [89, 114]. But because of the lack of memory isolation, the concurrent tasks are not scheduled in parallel.

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

Parallel programming is the only solution for efficiency, at the expense of development efforts to explicitely define the memory isolation of concurrent tasks and their communications by message pasing.

The table 3.5 presents a summary of the analysis of performance of the platforms presented in this section.

Model	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Event-driven programming	5	3		3
Lock-free Data-Structures	5	3		3
Multi-threading programming	4	3		3
Cooperative Threads	4	3		3

Table 3.5 – Efficiency of Concurrent Programming Platforms

3.3.1.2 Parallel Programming

Concurrent programming allows to define the tasks scheduling causally. Concurrent tasks can be scheduled in parallel only if their memory are isolated.

The Flynn's taxonomy [44] categorizes parallel executions in function of the multiplicity of their flow of instruction and data. Parallel programming models belong to the category Multiple Instruction Multiple Data (MIMD), which is further divided into Single Program Multiple Data (SPMD) [10, 32, 33] and Multiple Program Multiple Data (MPMD) [23, 21]. SPMD defines a single program replicated on many processing units [30, 82, 22] – it is derived from the multi-threading programming model presented in section **??**. While MPMD defines multiple parallel tasks in the implementation [56, 46, 45].

*

This section presents MPMD platforms allowing to define isolated tasks. It presents theoretical and programming models on asynchronous communication and

⚠ schema of SPMD and MPMD

isolated execution for parallel programming. It then presents stream processing programming models. And finally, it concludes on the limitations of parallel programming regarding productivity.

Theoretical Models The event-driven programming model used to cope with asynchronous communications allows the causal scheduling of concurrent tasks. This causal scheduling is sufficient to assure correctness in a distributed system [89, 114]. The Actor model allows to express the causal ordering of computation as a set of parallel actors communicating by asynchronous messages [70, 71, 27]. In reaction to a received message, an actor can create other actors, send messages, and choose how to respond to the next message. Additionally, the communication in reality are too slow compared to execution to be synchronous, and are subject to various faults and attacks [91]. The Actor model takes these physical limitations in account [72].

Similarly, coroutines are autonomous programs which communicate with adjacent modules as if they were input and output subroutines [29]. It defines a pipeline to implement multi-pass algorithms. Similar works include the Communicating Sequential Processes (CSP) [74, 17], and the Kahn Networks [86, 87].

3.3.1.3 Summary of Concurrent and Parallel Programming Models

Table 3.6 presents a summary of the analysis of the paradigm presented in the previous paragraphs.

Model	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Event-driven programming	5	3		3
Lock-free Data-Structures	5	3		3
Multi-threading programming	4	3		3
Cooperative Threads	4	3		3
Actor Model	5	5		5
Communicating Sequential Processes	5	5		5
Skeleton	4	4		4
Service Oriented Architecture	4	4		4
Microservices	4	4		4

Table 3.6 – Efficiency of Concurrent and Parallel Programming Platforms

3.3.2 Adoption

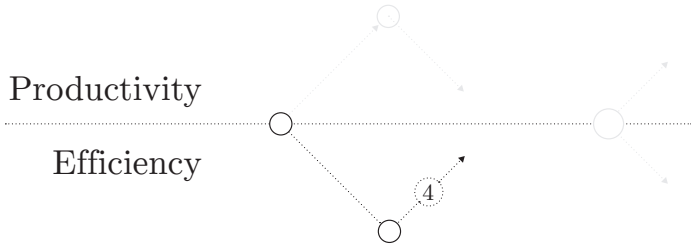


Figure 3.5 – Steering back toward Productivity

illustration:
mars rover

When the need for efficiency is higher than the need for productivity, the adoption is steered by the industry more than the community. If the industry really needs a platform, it will commit the required development effort despite a low productivity. The platforms for the Mars Rover or the banking systems are 30 years old, yet the industry continues to maintain them. The platform presented in this section emerged from the academia and the industry but are often barely known by the larger community of developers. The more the platform abandons productivity, the less it will be supported by the community.

3.3.2.1 Concurrent Programming

Most programming languages implementation supports concurrent programming somehow. Either with multi-threading or event-driven programming. These two are highly adopted by both the industry and the community, as presented in section 3.2.2.

On the other hand, lock-free data structures and cooperative threads comes from the academia, similarly to functionnal programming, and did not encounter significant adoption from the community.

Table 3.7 presents a summary of the adoption of concurrent programming models.

Model	Community support	Industrial need	→	Adoption
Event-driven programming	5	5		5
Lock-free Data-Structures	0	1		1
Multi-threading programming	3	5		5
Cooperative Threads	0	0		0

Table 3.7 – Adoption of Concurrent Programming Platforms

3.3.2.2 Parallel Programming

There exists several platforms directly inspired by the actors model, like Erlang [**JoeArmstrong**], Scala [107], Akka[16] and Play[17]. Scala is a programming language unifying the object model and functional programming. Akka is a framework based on Scala, following the actor model to build highly scalable and resilient applications. Play is a web framework based on top of Akka. And Erlang is a functional language designed by Ericsson to operate networks of telecommunication devices [**Armstrong2014**, 9, 103]

There are as well other platforms inspired by other theoretical model, like , inspired by Coroutines and CSP. Go is an open source language initiated by Google to build highly concurrent services.

These examples of implementation are largely used in the industry, but are almost unknown outside of it. They are backed by strong, but small passionated communities.

However, the organization in independent tasks is hardly compatible with the modular organization presented in the previous section. It is difficult for developers to manage the superposition of these two organizations, tasks and modules. This superposition makes these platforms accessible only to an elite in the industry supporting it. The next paragraphs present platforms mitigating the difficulty stemming from the duality between execution decomposition and modularity.

Tasks Organization and Communications To reduce the difficulties of the superposition of tasks and modules, algorithmic skeletons propose predefined patterns of organization to fit certain types of problems [28, 34, 99, 53]. Developers specialize a skeleton and focus on their problem independently of the required communication. These solutions are hardly used by the community, but are crucial in some industrial contexts. A famous example is the map/reduce pattern introduced by Google [34].

Tasks Granularity The Service Oriented Architectures (SOA) allows developers to express an application as an assembly of services connected to each others. Some examples of SOA platforms are OSGi[18], EJB[19] and Spring[20]. It allows to adjust the

[16] `http://akka.io/`
[17] `https://www.playframework.com/`
[18] `https://www.osgi.org/developer/specifications/`
[19] `http://www.oracle.com/technetwork/java/javaee/ejb/index.html`
[20] `http://projects.spring.io/spring-framework/`

granularity of tasks to help developers to better fit the tasks organization with the modular organization [2].

More recently, Microservices are tackling the same challenge on the web [42, 47, 102]. Some examples of Microservices are Seneca[21]. They 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.

The parallel programming platforms previously presented allow to build generic distributed systems. In the context of the web, a real-time application must process high volumes streams of requests within a certain time. The next paragraphs present platforms focusing on this challenge.

3.3.2.3 Stream Processing Systems

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 MPMD platforms presented until now. They borrows the syntax from SQL to run requests in parallel on continuous data streams. The computation of these requests spread over a distributed architecture. Some recent examples are DryadLINQ [80, 143], Apache Hive [129], Timestream [110], Shark [141].

Pipeline Architecture The pipeline architecture introduced by SEDA [138] organizes an application as a network of event-driven stages connected by explicit queues, the output of one feeding the input of the next. The event-driven paradigm of a stage is similar to work like Ninja [55] and Flash [108] previously presented. But the independence of stages allow to spread the execution on a parallel architecture. The academic works and industrial implementations of pipeline architecture are .

Parallel programming is barely supported by the community, but emerges mainly from industrial needs and academic research. The implementations improve efficiency, but prevent their adoption by the community due to a weak productivity. Despite the performance limitation, the event-driven programming model is the best candidate for a concurrent programming model supported by the community, and with concrete needs in the industry. Table 3.8 summarize the adoption of the platform oriented toward performance presented in this section.

[21] `http://senecajs.org/`

Model	Community support	Industrial need	→	Adoption
Event-driven programming	5	5		5
Lock-free Data-Structures	0	1		1
Multi-threading programming	3	5		5
Cooperative Threads	0	0		0
Actor Model	1	5		1
Communicating Sequential Processes	1	5		1
Skeleton	2	5		2
Service Oriented Architecture	3	4		3
Microservices	3	3		3

Table 3.8 – Adoption of Concurrent and Parallel Programming Platforms

3.3.3 Productivity Limitations

Parallel programming requires the organization of execution and memory into independent tasks. It allows the different granularity of state accessibility required for efficiency. At a fine level, the state is shared, while at a coarser level, it is isolated. This difference in state access impacts higher-order programming. It limits the composition of modules, hence impacts productivity.

Without good composition between modules, parallel programming forces to develop two mental representations – one for the module organization and one for the tasks organization – or to abandon the module organization and productivity altogether. It makes parallel programming productive only to an elite of developers that are able to keep the two mental representations.

This thesis focus on platforms allowing developers to be productive, and to produce efficient web applications to stimulate the economy. To fit the economical

context of this thesis, a solution must provide efficiency while avoiding the developers to keep a double mental representation of the implementation. It comes with an abstraction for the tasks and memory organization, for the developer to focus only on the module organization providing productivity. The next section presents some works that provides such an abstraction.

Model	Composition	Encapsulation	→	Productivity
Event-driven programming	5	5		5
Lock-free Data-Structures	5	5		5
Multi-threading programming	4	4		4
Cooperative Threads	4	4		4
Actor Model	2	2		2
Communicating Sequential Processes	2	2		2
Skeleton	2	2		2
Service Oriented Architecture	2	2		2
Microservices	2	2		2
Data Stream System Management	2	2		2
Pipeline Stream Processing	2	2		2

Table 3.9 – Productivity of Concurrent, Parallel and Stream Programming Platforms

3.3.4 Summary

Table 3.10 summarizes the characteristics of the platforms presented in this section.

Model	Productivity	Adoption	Efficiency
Event-driven programming	5	5	3
Lock-free Data-Structures	5	1	3
Multi-threading programming	4	5	3
Cooperative Threads	4	0	3
Actor Model	2	1	5
Communicating Sequential Processes	2	1	5
Skeleton	2	2	4
Service Oriented Architecture	2	3	4
Microservices	2	3	4
Data Stream System Management	2	1	3
Pipeline Stream Processing	2	2	5

Table 3.10 – Summary of Concurrent and Parallel Programming Platforms

3.4 Adoption Focused Platforms

Section 3.2 and section 3.3 present the platforms focusing respectively on productivity and efficiency, and conclude that favoring on one negatively impacts the other. Moreover, a balance between productivity and efficiency is required to be both supported by the community and needed by the industry, hence trigger a virtuous circle of adoption. This section presents platforms featuring an abstraction of the tasks organization to allow developers to focus on the modular organization to keep both productivity and efficiency. Section **??** presents Compilers, and section **??** presents Runtimes.

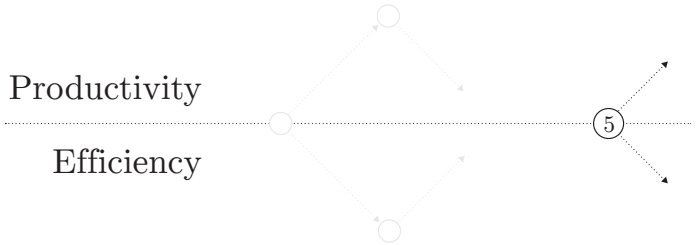


Figure 3.6 – Focus on Adoption

3.4.1 Abstraction of Tasks Organization

3.4.1.1 Compilers

"It is a mistake to attempt high concurrency without help from the compiler"
— R. Behren, J. Condit, E. Brewer [14]

.

As soon as the incompatibility between the modules and the tasks organizations were presented, it was suggested to use a compilation approach to mitigate this incompatibility [109]. This section presents the state of the art to extract parallelization from sequential programs through code transformation and compilation.

*

⚠ read and include [20]

Parallelism Extraction Extracting parallelism from a sequential implementation is a hard problem [84]. A compiler needs to identify the commutative operations to parallelize their executions [116, 26].

An important work was done to parallelize loop iterations [98, 5, 24, 11, 111], particularly using the polyhedral compilation method [12]. Examples of polyhedral compilers are . To improve performance gains outside of loops, some compilers identify the data-flow parallelism on the whole program [13, 20, 93]. Moreover, the data-flow representation and execution of a program is well suited for modern data processing applications [41], as well as web services [117].

Mutable closures required for higher-order programming remains a challenge to parallelize because of the memory references shared across the program [64, 105, 97]. The next paragraphs present some improvements in compilation applicable for parallelism extraction.

Static analysis Compilers statically analyze the control-flow of a program to detect commutative operations [4]. The point-to analysis identifies side-effects [7, 81, 120, 137] which allows to infer commutativity. However, this analysis is not sufficient to track the dynamic control-flow of higher-order functions [118] like used in Javascript.

Another approach, abstract interpretation, is to interpret the possible path of executions. It allows to statically reason on the behavior of dynamic program [**Raychev2013**, 94, 119, 50, 61, 49, 16]. It is successfully used for security applications [76, 85, 142, 95, 25, 40]*.

⚠ Update the citation for Dolby2015

However, these static analysis techniques remains often too imprecise, and expensive for the performance gain to be profitable. Instead, some compilers relies on annotations from the developers.

Annotations Some works proposed to rely on annotations from the developer to identify the shared data structures and infer the commutativity of operations [135, 41]. Such annotations are especially relevant for accelerators such as GPUs or FPGAs, because the development effort yields huge performance improvements [**Tarditi2006**]. Examples of such compilers are OpenMP [31], OpenCL [122], CUDA [106], Cg [96], Brook [18] and Liquid Metal [**Huang2008**].

Compilation Limitations For dynamic, higher-level languages like Javascript, the static analysis is not sufficient to correctly infer the independence of operations to parallelize them. And parallel compilers often fall back on relying on annotation provided by developers. Hence, the burden of detailing the tasks organization falls back to the developer, similarly to the platforms presented in the previous section.

Alternatively, another approach is to rely on the runtime to detect and distribute the commutative operations, and assure the communications. The next paragraphs present runtime allowing this dynamic distribution.

3.4.1.2 Runtimes

Partitioned Global Address Space The Partitioned Global Address Space (PGAS) provides a uniform memory access on a distributed architecture. It attempts to combine the efficiency of distributed memory systems, with the productivity 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 platform assures the remote accesses and synchronization of memory across nodes. Examples of implementation of the PGAS model are .

Dynamic Distribution of Execution Following SEDA, Leda proposes a model where the independent stages of the pipeline are defined only by their role in the application [**Salmito2014**, 117]. The execution distribution and module organization are different. The actual execution distribution is defined automatically during deployment. This automation manages the execution organizations to help the developer focus on the modular organization. However, it doesn't improve the composition of module with higher-order programming.

Tables 3.11 and 3.13 presents the platforms presented in this section regarding maintainability and performance.

Model	Composition	Encapsulation	→	Productivity
Partitionned Global Address Space	❷	❷		❷
Dynamic Distribution	❷	❷		❷
Polyhedral Compiler	❷	❷		❷
Annotation Compiler	❷	❷		❷

Table 3.11 – Productivity of Compilation and Runtime Platforms

Model	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Partitionned Global Address Space	4	4		4
Dynamic Distribution	4	4		4
Polyhedral Compiler	4	4		4
Annotation Compiler	4	4		4

Table 3.12 – Efficiency of Compilation and Runtime Platforms

3.4.2 Adoption Limitations

All the platforms presented in this section come from the need of the industry to reduce the development commitment required for efficiency. However, these platforms are limited to scientific applications. They respond exclusively to academic or industrial needs, and are barely supported by the community.

The balance between efficiency and productivity is not sufficient for a community of passionate to gather around the platform. The platforms need to answer to needs of small scale for novice to start learning, and to incite the community to experiment and start projects organically. The context of web development is particularly adapted for this requirement.

Model	Community support	Industrial need	→	Adoption
Partitionned Global Address Space	0	3		0
Dynamic Distribution	0	3		0
Polyhedral Compiler	0	3		0
Annotation Compiler	0	3		0

Table 3.13 – Adoption of Compilation and Runtime Platforms

3.4.3 Summary

Table 3.14 summarizes the characteristics of the platforms presented in this section.

Model	Productivity	Adoption	Efficiency
Partitionned Global Address Space	2	0	4
Dynamic Distribution	2	0	4
Polyhedral Compiler	2	0	4
Annotation Compiler	2	0	4

Table 3.14 – Summary of Compilation and Runtime Platforms

3.5 Analysis

This chapter presented a broad view of platforms and their balance between productivity or efficiency. The platforms favoring one sacrifice the other. The adoption, and usage of these platforms prove that none of these compromises are sustainable.

The productive platforms are highly adopted. Their productivity feed a reinforcing circle of adoption between the community and the industry. However, they lack the efficiency required to strive in the latter stage of a project, where efficiency becomes crucial.

On the other hand, the efficient platforms are not widely adopted by the community. These platforms are unable to respond to the need of the community to prototype and to experiment on small projects to make them evolve into larger ones later on.

Continuous Development It is not possible for a platform to support both productivity and efficiency at the same time. These platforms are oriented toward productivity, efficiency or a compromise between both. As the two are required at different time in the evolution of the project, to follow a project, a platform need to meet the requirements at the good time. They need to supporting productivity to allow the community to experiment, and organically start projects. And then continuously shift toward efficiency as the project evolve, and requires it.

None of these platforms are able to support productivity then efficiency to follow the evolution of a project. They lack the possibility to make a project evolve from the very early stage until maturation. A project needs to change platform to change the priority. These shifts of platforms have economical consequences.

The table 3.15 summarizes the analysis of the state of the art presented in this chapter.

Model	Productivity	Adoption	Efficiency
Imperative Programming	3	3	0
Object-Oriented Programming	5	4	0

Functional Programming	5	1	0
Multi Paradigm	5	4	0
Event-driven programming	5	5	3
Lock-free Data-Structures	5	1	3
Multi-threading programming	4	5	3
Cooperative Threads	4	0	3
Actor Model	2	1	5
Communicating Sequential Processes	2	1	5
Skeleton	2	2	4
Service Oriented Architecture	2	3	4
Microservices	2	3	4
Data Stream System Management	2	1	3
Pipeline Stream Processing	2	2	5
Partitionned Global Address Space	2	0	4
Dynamic Distribution	2	0	4
Polyhedral Compiler	2	0	4
Annotation Compiler	2	0	4

Table 3.15 – Summary of the state of the art

Chapter 4

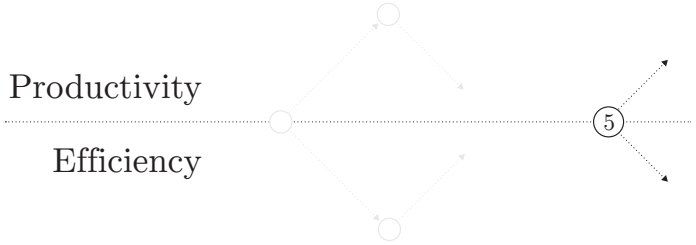
Pipeline parallelism for Javascript

The conclusion of the state of the art is that no platform allows to follow a project from the early beginning to the maturation of the project. Indeed, no platform can provide alternatively productivity and efficiency. All the platform tends to focus on a static compromise between these two goals, and therefore, are useful only at a precise point in the project. They either grow useless, or are too complicated to begin with.

This chapter presents the solution developped in this thesis. A platform to follow web application projects from the early beginning until the maturation of the project.

To support this evolution, it support a continuous developpement.

4.1 Seamless Development



The section **??** 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 **??** shows that to attain performance scalability, an application needs to multiply the exclusive accesses to its state. That implies to 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 **??**, 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

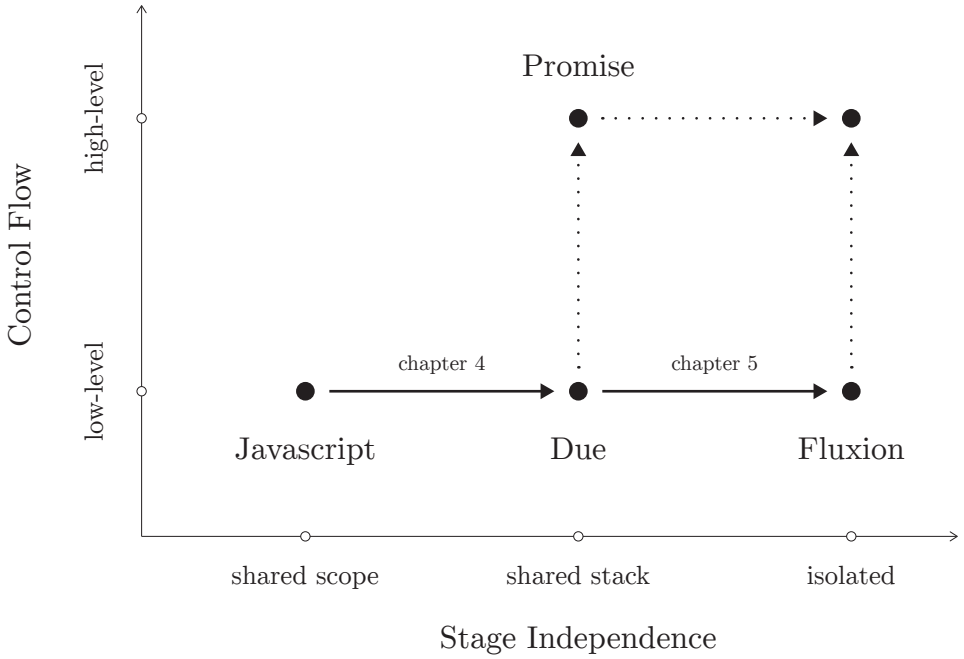


Figure 4.1 – Roadmap

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.

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 [101].

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 [137], or a program logic [49, 16]. 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).

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

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

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 **??**.

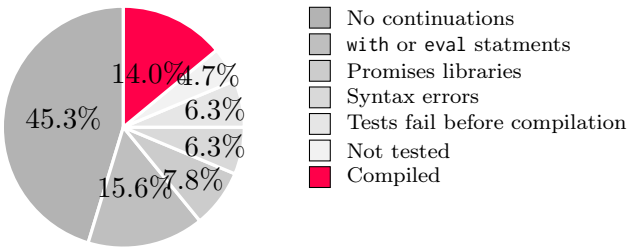


Figure 5.1 – Compilation results distribution

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.

```
var app = require('express')(),
    fs = require('fs'),
    count = 0;

app.get('/', function handler(req, res){
  fs.readFile(__filename, function reply(err, data) {
    count += 1;
    res.send(err || template(count, data));
  });
});

app.listen(8080);
```

Listing 6.1 – Example web application

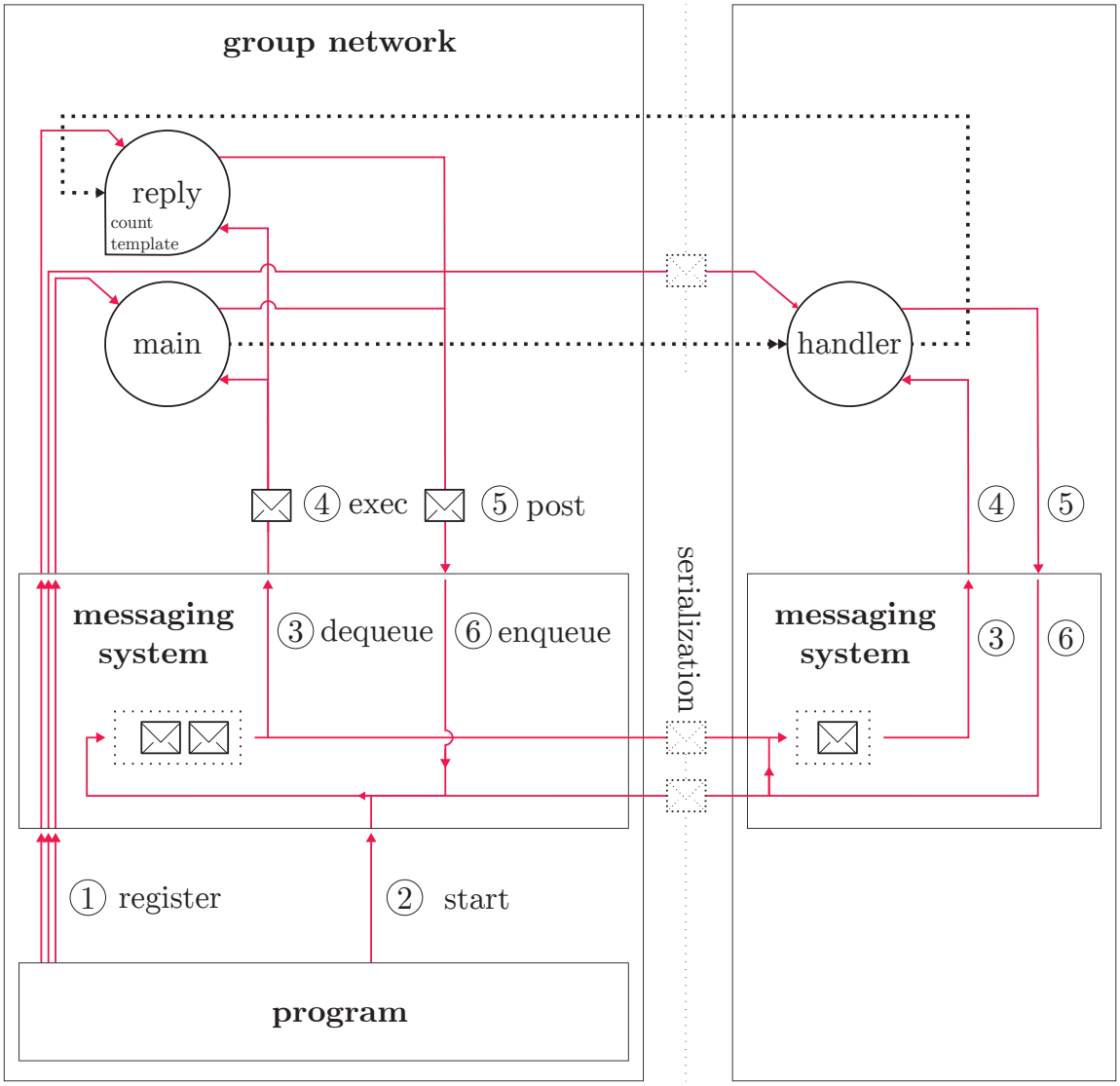


Figure 6.2 – The fluxionnal execution model in details

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.

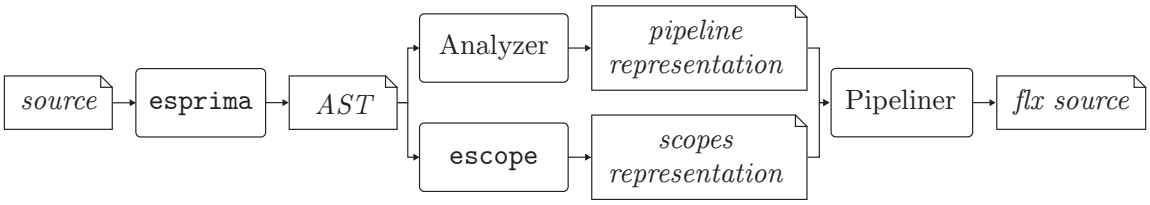


Figure 6.3 – Compilation chain

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.

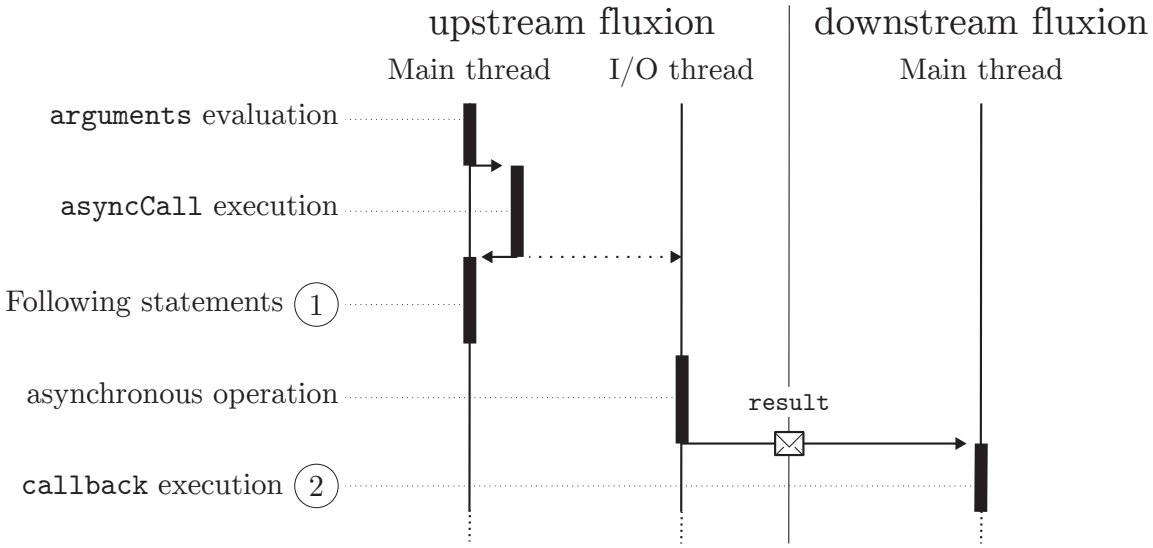
6.2.1 Analyzer step

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.

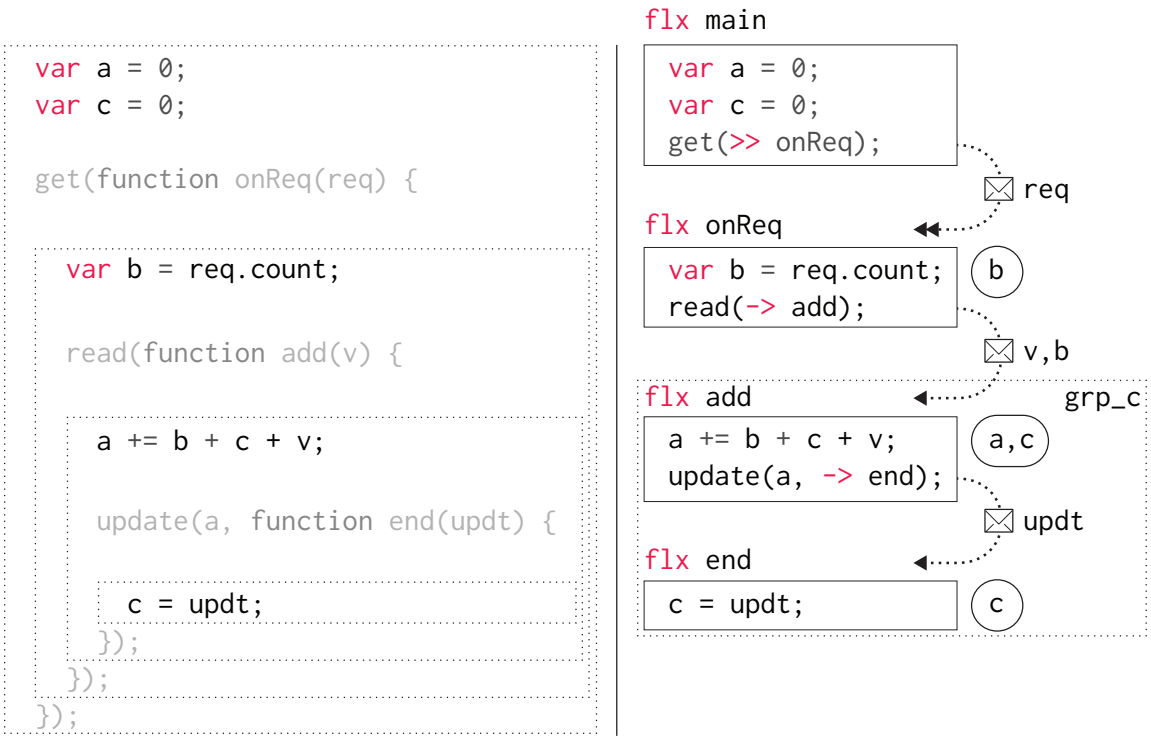


Figure 6.5 – Variable management from Javascript to the high-level fluxionnal language

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]

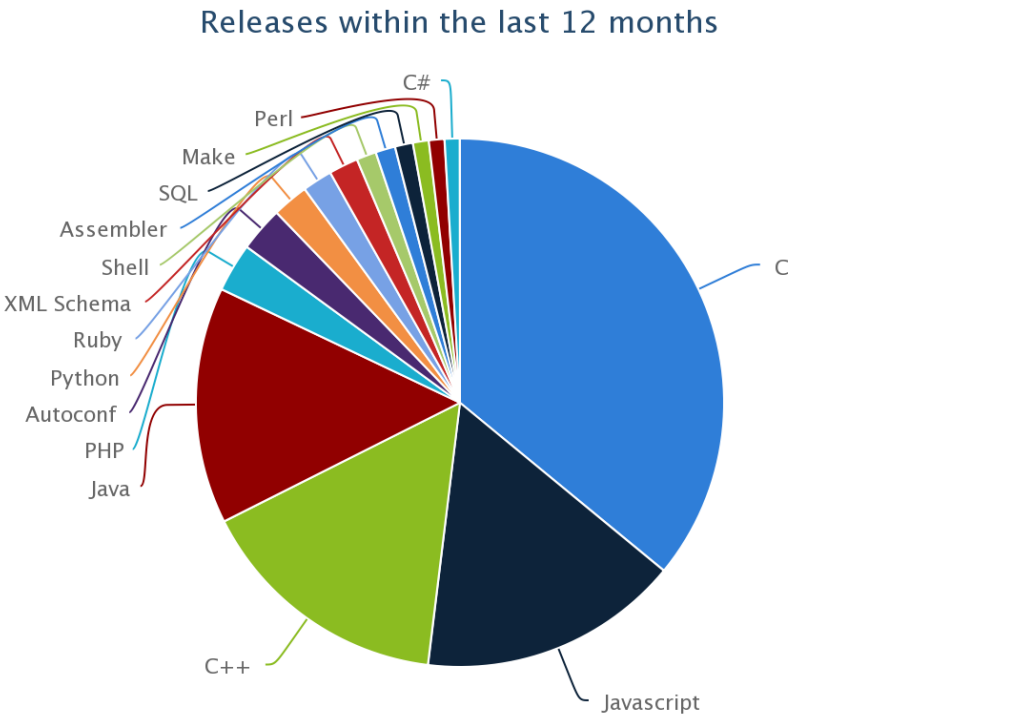
The black-duck, which analyze the usage of language on many forges, and collaborative hosts, rank Javascript number 2, after C, and with about the same usage as C++.

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] H Abelson, G J Sussman, and J Sussman. *The Structure and Interpretation of Computer Programs*. Vol. 9. 3. 1985, p. 81. DOI: 10.2307/3679579.

[2] Sebastian Adam and Joerg Doerr. "How to better align BPM & SOA - Ideas on improving the transition between process design and deployment". In: *CEUR Workshop Proceedings*. Vol. 335. 2008, pp. 49–55.

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

[4] Frances E. Allen. "Control flow analysis". In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. DOI: 10.1145/390013.808479.

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

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

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

[8] James H. Anderson and Mohamed G. Gouda. *The virtue of Patience: Concurrent Programming With And Without Waiting*. 1990.

[9] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in ERLANG*. 1993.

[10] Michel Auguin and Francois Larbey. "OPSILA: an advanced SIMD for numerical analysis and signal processing". In: *Microcomputers: developments in industry, business, and education*. 1983, pp. 311–318.

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

[12] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting Polyhedral Loop Transformations to Work". In: *LCPC '04 Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science 2958.Chapter 14 (2004). Ed. by Lawrence Rauchwerger, pp. 209–225. DOI: 10.1007/b95707.

[13] Micah Beck, Richard Johnson, and Keshav Pingali. "From control flow to dataflow". In: *Journal of Parallel and Distributed Computing* 12.2 (1991), pp. 118–129. DOI: 10.1016/0743-7315(91)90016-3.

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

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

[16] 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).

[17] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. "A Theory of Communicating Sequential Processes". In: *Journal of the ACM* 31.3 (June 1984), pp. 560–599. DOI: 10.1145/828.833.

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

[19] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. "Identification of coordination requirements". In: *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work - CSCW '06*. New York, New York, USA: ACM Press, Nov. 2006, p. 353. DOI: 10.1145/1180875.1180929.

[20] Bryan Catanzaro, Shoaib Kamil, and Yunsup Lee. "SEJITS: Getting productivity and performance with selective embedded JIT specialization". In: *... Models for Emerging ...* (2009), pp. 1–10. DOI: 10.1.1.212.6088.

[21] F Chan, J N Cao, A T S Chan, and M Y Guo. "Programming support for MPMD parallel computing in ClusterGOP". In: *IEICE Transactions on Information and Systems* E87D.7 (2004), pp. 1693–1702.

[22] K. Mani Chandy and Carl Kesselman. "Compositional C++: Compositional parallel programming". In: *Languages and Compilers for Parallel Computing*. Vol. 757. 2005, pp. 124–144. DOI: 10.1007/3-540-48319-5.

[23] Chi-Chao Chang, G. Czajkowski, T. Von Eicken, and C. Kesselman. "Evaluating the Performance Limitations of MPMD Communication". In: *ACM/IEEE SC 1997 Conference (SC'97)* (1997), pp. 1–10. DOI: 10.1109/SC.1997.10040.

[24] Chun Chen, Jacqueline Chame, and Mary Hall. "CHiLL: A framework for composing high-level loop transformations". In: *U. of Southern California, Tech. Rep* (2008), pp. 1–28. DOI: 10.1001/archneur.64.6.785.

[25] Andrey Chudnov and David A. Naumann. "Inlined Information Flow Monitoring for JavaScript". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Oct. 2015), pp. 629–643. DOI: 10.1145/2810103.2813684.

[26] 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.

[27] William Douglas Clinger. "Foundations of Actor Semantics". eng. In: (May 1981).

[28] M. I. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation*. eng. 1988.

[29] 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.

[30] David E. Culler, A. Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. "Parallel programming in Split-C". English. In: (), pp. 262–273. DOI: 10.1109/SUPERC.1993.1263470.

[31] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". English. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[32] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. "A single-program-multiple-data computational model for EPEX/FORTRAN". In: *Parallel Computing* 7.1 (Apr. 1988), pp. 11–24. DOI: 10.1016/0167-8191(88)90094-4.

[33] Frederica Darema. "The SPMD Model: Past , Present and Future". In: *Parallel Computing*. 2001, p. 1. DOI: 10.1007/3-540-45417-9{_}1.

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

[35] E W Dijkstra. *Notes on structured programming*. 1970.

[36] Edsger Dijkstra. "Over de sequentialiteit van procesbeschrijvingen". In: ().

[37] 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.

[38] 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.

[39] 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.

[40] Julian Dolby. "A History of JavaScript Static Analysis with WALA at IBM". In: (2015).

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

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

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

[44] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". English. In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[45] Ian Foster, Carl Kesselman, and Steven Tuecke. "The Nexus Approach to Integrating Multithreading and Communication". In: *Journal of Parallel and Distributed Computing* 37.1 (Aug. 1996), pp. 70–82. DOI: 10.1006/jpdc.1996.0108.

[46] I.T. Foster and K M Chandy. "Fortran M: A Language for Modular Parallel Programming". In: *Journal of Parallel and Distributed Computing* 26.1 (Apr. 1995), pp. 24–35. DOI: 10.1006/jpdc.1995.1044.

[47] M Fowler and J Lewis. "Microservices". In: *. . . . http://martinfowler. com/articles/microservices. html [. . .* (2014).

[48] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: *ACM SIGPLAN Notices* 33.5 (May 1998), pp. 212–223. DOI: 10.1145/277652.277725. arXiv: 9809069v1 [arXiv:gr-qc].

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

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

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

[52] Adele Goldberg. *Smalltalk-80 : the interactive programming environment*. 1984, xi, 516 p.

[53] Horacio González-Vélez and Mario Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". In: *Software: Practice and Experience* 40.12 (Nov. 2010), pp. 1135–1160. DOI: 10.1002/spe.1026.

[54] J Gosling. *The Java language specification*. 2000.

[55] Steven D. Gribble, Matt Welsh, Rob Von Behren, Eric a. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. "Ninja architecture for robust Internet-scale systems and services". In: *Computer Networks* 35.4 (2001), pp. 473–497. DOI: 10.1016/S1389-1286(00)00179-1.

[56] Andrew S. Grimshaw. "An Introduction to Parallel Object-Oriented Programming with Mentat". In: (Apr. 1991).

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

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

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

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

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

[62] P.B. Hansen and J. Staunstrup. “Specification and Implementation of Mutual Exclusion”. English. In: *IEEE Transactions on Software Engineering* SE-4.5 (Sept. 1978), pp. 365–370. DOI: 10.1109/TSE.1978.233856.

[63] Tim Harris, James Larus, and Ravi Rajwar. “Transactional Memory, 2nd edition”. en. In: *Synthesis Lectures on Computer Architecture* 5.1 (Dec. 2010), pp. 1–263. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[64] Williams Ludwell Harrison. “The interprocedural analysis and automatic parallelization of Scheme programs”. In: *Lisp and Symbolic Computation* 2.3-4 (Oct. 1989), pp. 179–396. DOI: 10.1007/BF01808954.

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

[66] Danny Hendler, Nir Shavit, and Lena Yerushalmi. “A scalable lock-free stack algorithm”. In: *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA '04*. New York, New York, USA: ACM Press, June 2004, p. 206. DOI: 10.1145/1007912.1007944.

[67] M. Herlihy. “A methodology for implementing highly concurrent data structures”. In: *ACM SIGPLAN Notices* 25.3 (Mar. 1990), pp. 197–206. DOI: 10.1145/99164.99185.

[68] Maurice Herlihy. “Wait-free synchronization”. In: *ACM Transactions on Programming Languages and Systems* 13.1 (Jan. 1991), pp. 124–149. DOI: 10.1145/114005.102808.

[69] Maurice P. Herlihy. “Impossibility and universality results for wait-free synchronization”. In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing - PODC '88*. New York, New York, USA: ACM Press, Jan. 1988, pp. 276–290. DOI: 10.1145/62546.62593.

[70] 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).

[71] Carl Hewitt. “Viewing control structures as patterns of passing messages”. In: *Artificial intelligence* (1977).

[72] Carl Hewitt and Jr Baker Henry. “Actors and Continuous Functionals,” in: (Dec. 1977).

[73] Martin Hilbert and Priscila López. "The world's technological capacity to store, communicate, and compute information." In: *Science (New York, N.Y.)* 332.6025 (Apr. 2011), pp. 60–65. DOI: 10.1126/science.1200970.

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

[75] 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.

[76] 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).

[77] Paul Hudak, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, and John Hughes. "Report on the programming language Haskell". In: *ACM SIGPLAN Notices* 27.5 (May 1992), pp. 1–164. DOI: 10.1145/130697.130699.

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

[79] Walter Hürsch and Cristina Videira Lopes. *Separation of Concerns*. Tech. rep. NU-CCS-95-03. 1995.

[80] 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).

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

[82] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. "CRL: High-Performance All-Software Distributed Shared Memory". In: *ACM SIGOPS Operating Systems Review* 29.5 (Dec. 1995), pp. 213–226. DOI: 10.1145/224057.224073.

[83] Ralph E. Johnson and Brian Foote. "Designing Reusable Classes Abstract Designing Reusable Classes". In: *Journal of Object-Oriented Programming* 1 (1988), pp. 22–35.

[84] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in dataflow programming languages". In: *ACM Computing Surveys* 36.1 (Mar. 2004), pp. 1–34. DOI: 10.1145/1013208.1013209.

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

[86] Gilles Kahn. "The semantics of a simple language for parallel programming". In: *In Information Processing'74: Proceedings of the IFIP Congress* 74 (1974), pp. 471–475.

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

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

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

[90] Leslie Lamport. "Concurrent reading and writing". In: *Communications of the ACM* 20.11 (Nov. 1977), pp. 806–811. DOI: 10.1145/359863.359878.

[91] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pp. 382–401. DOI: 10.1145/357172.357176.

[92] Charles E. Leiserson. "The Cilk++ concurrency platform". In: *Journal of Supercomputing* 51.3 (Mar. 2010), pp. 244–257. DOI: 10.1007/s11227-010-0405-3.

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

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

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

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

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

[98] Christophe Mauras. "Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones". PhD thesis. Jan. 1989.

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

[100] R. Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. 1997, p. 128.

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

[102] Dmitry Namiot and Manfred Sneps-Sneppe. *On Micro-services Architecture*. en. Aug. 2014.

[103] Jay Nelson. "Structured programming using processes". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang - ERLANG '04*. New York, New York, USA: ACM Press, Sept. 2004, pp. 54–64. DOI: 10.1145/1022471.1022480.

[104] R Nelson. "Including queueing effects in Amdahl's law". In: *Communications of the ACM* (1996).

[105] Jens Nicolay. "Automatic Parallelization of Scheme Programs using Static Analysis". PhD thesis. 2010.

[106] C Nvidia. "Compute unified device architecture programming guide". In: (2007).

[107] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. "An Overview of the Scala Programming Language". In: *System* Section 2 (2004), pp. 1–130.

[108] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. *Flash : An Efficient and Portable Web Server*. 1999. DOI: 10.1.1.119.6738.

[109] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058. DOI: 10.1145/361598.361623.

[110] 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).

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

[112] Arunmoezhi Ramachandran and Neeraj Mittal. “A Fast Lock-Free Internal Binary Search Tree”. In: *Proceedings of the 2015 International Conference on Distributed Computing and Networking - ICDCN ’15*. New York, New York, USA: ACM Press, Jan. 2015, pp. 1–10. DOI: 10.1145/2684464.2684472.

[113] K.H. Randall. “Cilk: Efficient Multithreaded Computing”. PhD thesis. 1998.

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

[115] J Rees and W Clinger. “Revised report on the algorithmic language scheme”. In: *ACM SIGPLAN Notices* 21.12 (Dec. 1986), pp. 37–79. DOI: 10.1145/15042.15043.

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

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

[118] O. Shivers. “Control-flow analysis of higher-order languages”. PhD thesis. 1991, pp. 1–186.

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

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

[121] 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.

[122] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[123] B Stroustrup. “The C++ programming language”. In: (1986).

[124] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. “The structure and value of modularity in software design”. In: *ACM SIGSOFT Software Engineering Notes* 26.5 (Sept. 2001), p. 99. DOI: 10.1145/503271.503224.

[125] H. Sundell and P. Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Proceedings International Parallel and Distributed Processing Symposium* 00.C (2003), p. 11. DOI: 10.1109/IPDPS.2003.1213189.

[126] Gerald Jay Sussman and Jr Steele, Guy L. "Scheme: A interpreter for extended lambda calculus". In: *Higher-Order and Symbolic Computation* 11 (1998), pp. 405–439. DOI: 10.1023/A:1010035624696.

[127] Richard E Sweet. "The Mesa programming environment". In: *ACM SIGPLAN Notices*. Vol. 20. 7. 1985, pp. 216–229. DOI: 10.1145/17919.806843.

[128] P. Tarr, H. Ossher, W. Harrison, and Jr. Sutton, S.M. "N degrees of separation: multi-dimensional separation of concerns". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)* (1999), pp. 107–119. DOI: 10.1145/302405.302457.

[129] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1626–1629. DOI: 10.14778/1687553.1687609.

[130] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. "Wait-free linked-lists". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7702 LNCS. 2012, pp. 330–344. DOI: 10.1007/978-3-642-35476-2{_}23.

[131] D Turner. "An overview of Miranda". In: *ACM SIGPLAN Notices* 21.12 (Dec. 1986), pp. 158–166. DOI: 10.1145/15042.15053.

[132] D. A. Turner. "The semantic elegance of applicative languages". In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture - FPCA '81*. New York, New York, USA: ACM Press, Oct. 1981, pp. 85–92. DOI: 10.1145/800223.806766.

[133] John D. Valois. "Lock-free linked lists using compare-and-swap". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95*. New York, New York, USA: ACM Press, Aug. 1995, pp. 214–222. DOI: 10.1145/224964.224988.

[134] Peter Van Roy and Seif Haridi. "Concepts, Techniques, and Models of Computer Programming". In: *Theory and Practice of Logic Programming* 5 (2003), pp. 595–600. DOI: 10.1017/S1471068405002450.

[135] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. "The Paralax infrastructure: automatic parallelization with a helping hand". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, New York, USA: ACM Press, Sept. 2010, pp. 389–399. DOI: 10.1145/1854273.1854322.

[136] Philip Wadler. "The essence of functional programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*. New York, New York, USA: ACM Press, Feb. 1992, pp. 1–14. DOI: 10.1145/143165.143169.

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

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

[139] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. "The lock-free k-LSM relaxed priority queue". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP 2015*. New York, New York, USA: ACM Press, Jan. 2015, pp. 277–278. DOI: 10.1145/2688500.2688547.

[140] 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.

[141] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Shark". In: *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*. New York, New York, USA: ACM Press, June 2013, p. 13. DOI: 10.1145/2463676.2465288.

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

[143] 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).