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

Etienne Brodu

January 19, 2016

Abstract

TODO translate from below when ready

Résumé

Internet démultiplie nos moyens de communications. Tout en réduisant leur latence de manière à développer l'économie à l'échelle planétaire. Il permet 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 promesse 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. Ce 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 parce qu'ils intégrent cette approche et permettent d'intégrer facilement de nouvelles fonctionnalités.

Une application qui répond correctement aux besoins atteindra de manière virale un nombre important d'utilisateurs. Son audience peut prendre plusieurs ordres de grandeurs en quelques jours voire en quelques heures si elle est correctement relayée. Une application est dite *scalable* si elle peut absorber ces augmentations d'audience. Or il est difficile pour une application d'être à la fois modulaire et *scalable*.

Au moment où l'audience devient très importante, il est souvent 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. Cela représentent une charge de travail conséquente et incertaine. De plus, l'équipe de développement doit concilier cette nouvelle approche de développement *scalable*, avec la demande en fonctionnalités. Aucun langage ne concilie ces deux objectifs. La maitrise de ces enjeux est clé pour la pérennité de l'application.

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 favorise les bonnes pratiques de développement pour une meilleure maintenabilité. La seconde propose 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 développement 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 7 years old I was laid amazed 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 productivity and performance efficiency, in the case of real-time web applications*.

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 application, allows the development team to focus on application specifications rather than technical optimizations and ease maintenance. The purpose of this PhD work, was to separate development productivity from performance efficiency, 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 applications.

1.1 Web development

Internet allows 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 a 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] Often the development team quickly concretises an MVP and iterates on it using a feature-driven and monolithic approach thanks to imperative languages like Java or Ruby.

1.2 Performance requirements

If the application successfully complies with users requirements, its user base might grow with its popularity. The application is scalable when it can efficiently respond to this growth. However, it is difficult to develop scalable applications with the feature-driven approach mentioned above. Eventually this growth requires to discard the

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

initial monolithic approach to adopt a more efficient processing model instead. Many of the most efficient models distribute the application on a cluster of commodity machines.

Once split, the application 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? It implies the reconciliation between the productivity required in the early stage of development and the efficiency required with the growth of popularity. To answer this question, this thesis proposes a solution based on the equivalence between two different programming models. On one hand, there is the asynchronous, functional programming model, embodied by the Javascript event-loop. On the other hand, there is the distributed, dataflow programming model, embodied by the the pipeline architecture.

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, enforcing isolation between the stages of this pipeline. With these two contributions, it is possible to transform the modular representation of an applicaiton into a pipeline representation. The modular representation allows develoment productivity, while the pipeline representation carries its execution efficiently. A development team shall then use these two representations to continuously iterate over the implementation of an application, and reach the best compromise between productivity and efficiency.

1.4 Thesis organization

This thesis is organized in six 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 chapter 2 to help the reader understand better the context. The end of this chapter presents clearly the problematic addressed in this thesis. Chapter 4 introduces the proposition of this thesis, and the articulation of the contributions. Chapter 5 presents the implementations of the two contributions. The first contribution allows to represent a program as a pipeline of stages. It introduces Dues, based on Javascript Promises, to layout the pipeline. The second contribution allows to make these stages independent. It introduces Fluxions to isolate these stages, and distribute the execution. Chapter 6 evaluates this proposition at the light of the previous works, and draws the possible perspectives beyond this work. Finally, chapter 7 concludes this thesis.

Chapter 2

Context and objectives

Contents

The web allows applications to grow a user base very quickly. The developement of a web application needs to follow this rapid pace to assure performance efficiency. But 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 defines the scope of this thesis. Section 2.1 presents the motivations that led the web to become a software platform, and the context of web development. It presents Javascript, one of the most important language in 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 application 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 applications, 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.

2.1.1 The Language of the Web

In the 80's, reducing development time became more profitable than reducing hardware costs. Higher-level languages replaced lower-level languages. The economical gain in development time brought by productive languages 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 a community became critical for the adoption, evolution and maturation of a language. Communities adopts a language because it allows to quickly experiment and enter businesses sectors. The industry adopts a language because it responds to business needs and its community represents a hiring pool. The community support and the industrial needs are reinforcing each other in a loop.

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. The Wordpress publishing platform is another example of an economical success with PHP.

The web acts as a catalyst in the interaction between the community and the industry. Because of its position in the web, 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 differences between the two implementations forced Web pages to be designed for a specific browser. This competition was fragmenting the Web. 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.

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.

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

But this situation evolved drastically since. All web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime [47]. This position became an incentive to make it fast (V8, ASM.js) and convenient (ES6, ES7). Any Javascript code in the browser is open, allowing the community to pick, improve and reproduce the best techniques [2]. Javascript is distributed freely, with all the tools needed to reproduce and experiment on the largest communication network in history. And since 2009, it came back on the server[3] 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[4]

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

ECMA International allowed 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 [55]. 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[5] and DOJO[6] are early famous examples, and

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

[3] True hipsters used Server-Side Javascript before it was cool. `https://developer.mozilla.org/en-US/docs/Archive/Web/Server-Side_JavaScript`

[4] `http://javascript.crockford.com/survey.html`

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

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

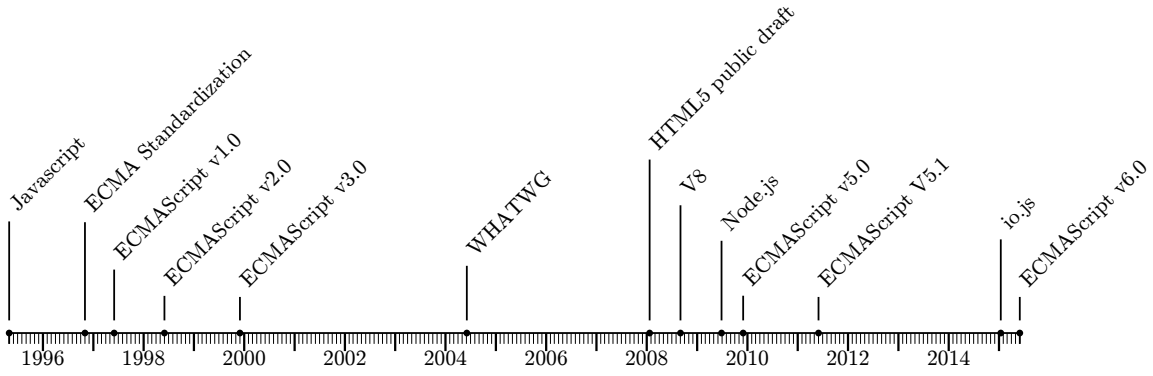


Figure 2.1: Javascript timeline

later jQuery[7] and underscore[8].

Since 2004, the Web Hypertext Application Technology Working Group[9] 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[10]. The milestones in the history of Javascript presented in the previous chapters are summariezed in figure 2.1.1.2.

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 applications against billions of users.

[7] `https://jquery.com/`
[8] `http://underscorejs.org/`
[9] `https://whatwg.org/`
[10] `http://blog.codinghorror.com/javascript-the-lingua-franca-of-the-web/`

2.1.2 Highly Concurrent Web Servers

Since the web allowed an application to scale world wide with near zero latency, the software industry needed innovative solutions to cope with large network traffic.

The Internet allows communication at an unprecedented scale. There is more than 16 billions connected devices, and it is growing fast[11] [78]. A large web application like google search receives about 40,000 requests per seconds[12]. 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[13]. In the 2010s, the limit is set at 10 millions simultaneous connections[14]. With the growing number of connected devices on the internet, concurrency is a very important property in the design of web applications.

2.1.2.1 Event-driven Execution Model

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

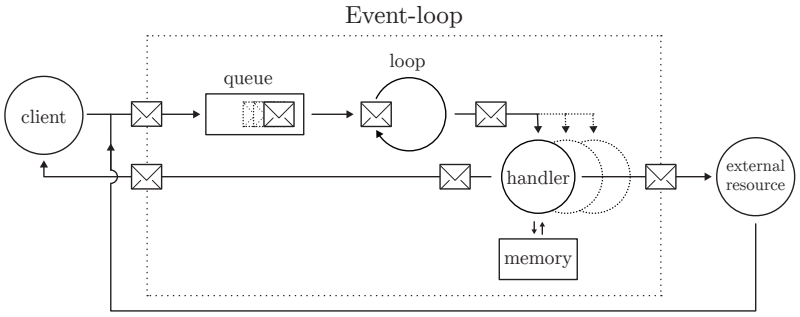


Figure 2.2: Event-driven execution model

The event-driven execution model is presented in figure 2.1.2.2. At reception, each request from a client queues an event waiting to be processed. A loop unqueues these events one at a time, and run the appropriate handler to process them. To process

[11] http://blogs.cisco.com/news/cisco-connections-counter
[12] http://www.internetlivestats.com/google-search-statistics/
[13] http://www.kegel.com/c10k.html
[14] http://c10m.robertgraham.com/p/manifesto.html

an event, a handler can query remote resources, which respond asynchronously by queueing additional events, processed by new handlers. Alternatively, a handler can respond directly to the client, ending this chain of asynchronous events.

This execution model allows the high concurrency required 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, while the traffic of a less popular web application is much more uncertain. Moreover, the load of the web application grows with its user base. The available resources need to increase as well to meet this load. For stable traffic, this growth is steady enough to plan the increase of resources ahead of time. But for unstable traffic, it is erratic and challenging to meet.

An application is scalable, if it is able to spread over resources proportionally as a reaction to its load to use these resources efficiently. It is a desirable property, as it helps to meet the growth, without spending time to manually spread the application on available resources to react to this erratic growth.

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-driven model and the pipeline execution model.

2.1.2.2 Pipeline Execution Model

The pipeline execution model, presented in figure 2.1.2.2, 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.

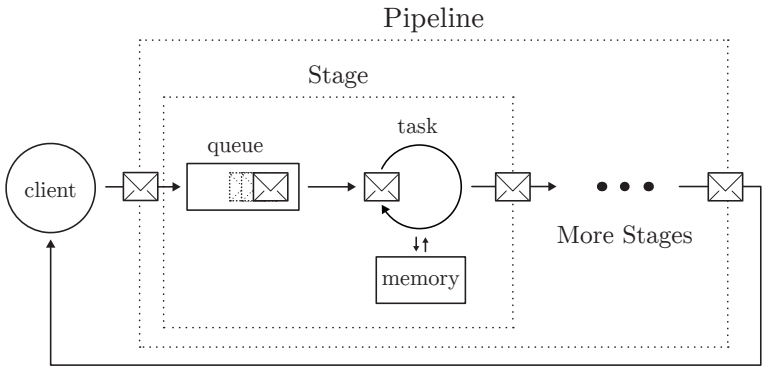


Figure 2.3: Pipeline executin model

The execution model of each stage is organized in a similar fashion than the event-loop presented previously. It receives and queues messages from upstream stages, processes them one after the other, and outputs the result to downstream stages. The pipeline architecture is different as each task is executed on an isolated stage. Whereas in the event-driven execution model, all handlers share the same queue, loop and memory store. This difference is illustrated in figure 2.1.2.2. The isolation of memory in the pipeline execution model impacts the productivity of its programming model. The next section details further the incompatibility between the two programming model and the resulting economical consequences.

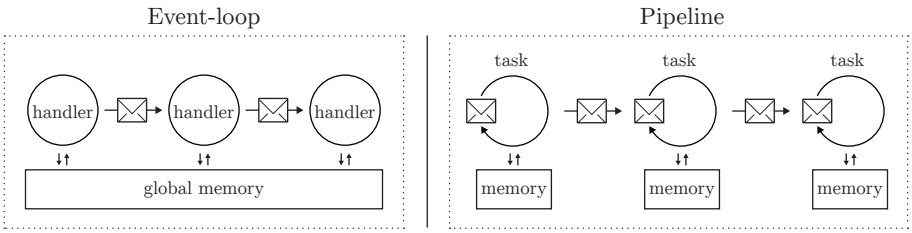


Figure 2.4: Comparison of the two memory models

2.2 An Economical Problem

With SaaS, the software industry is in charge of both development and 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 Web Development

The economical constraints to meet are very different in the beginning and during the maturation of a web application. In the early steps the constraints hold on the development productivity. The team needs to reduce development costs, and to release a first version as soon as possible. On the contrary, during the maturation of the application, the constraints hold on the performance efficiency. The application needs to be highly concurrent to meet the load of usage.

The team need to revise its approach to meet these different constraints. These revisions leads to disruptions in the evolution of the application.

2.2.1.1 Power-Wall Disruption

Around 2004, manufacturers reached what they called the *Power-wall*. The speed of sequential execution on a processing unit plateaued[15]. Therefore, the performance of sequential programming plateaued as well. They started to arrange transistors into several processing units to keep increasing overall performance efficiency. Parallel programming became the only option to achieve high concurrency, but the memory isolation it requires limits the productivity. This *Power-wall* leads to a rupture between efficiency and productivity.

2.2.1.2 Unavoidable Modularity

The best practices for productivity 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

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

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

The development team opt for a popular and accessible language to be productive in the beginning of the project. It is only after a certain threshold of user load that the economical constraint on efficiency exceeds the one on productivity. 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 proposition of this thesis is presented in the next section. The proposed solution 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.* This thesis intends to allow the developer to focus solely on business logic, and leave the technical constraints of performance scalability to automated tools. The objective of this work are avoid the disruption in development, and provide a seamless development experience. They are presented in the next paragraphs.

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.

As presented in the previous section, such applications are often implemented with the event-driven programming model or the pipeline programming model. Despite the differences between the two models, an equivalence to map these differences is developed throughout this thesis.

2.2.2.2 Differences

Both programmng models encapsulate the execution in tasks assured to have an exclusive access to the memory. However, they use two different models to provide this exclusivity. 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 chapter 4 and 5.

2.2.2.3 Equivalence

In the beginning of a project, the team adopts the event-driven 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-driven 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 [133]

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.

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, Productivity, Efficiency and Adoption, as well as all the underlying concepts.

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 [130], which helps developers to repair and enhance the application. Additionally, it reduces development time by allowing several developers to simultaneously implement different modules [151, 22].

The criteria to define modules to improve productivity are high cohesion enforced by enccapsulation and low coupling enforced by composition [130]. Cohesion defines how strongly the features inside a module are related. Coupling defines the strength of the interdependences between modules.

In this thesis, the criteria retained for productivity are the encapsulation and composition allowed by a platform. Encapsulation relies on the definition of boundaries, and the protection of data. Composition relies on higher-order programming and lazy evaluation. The next paragraphs define these requirements.

- Encapsulation (Boundary definition, Data protection)

  → increases Cohesion

- Composition (Higher-order programming / Lambda Expressions, Lazy evaluation / Stream composition)

  → decreases Coupling

3.1.1.2 Encapsulation

Boundary Definition Modular Programming draws clear interfaces around a piece of implementation so that the execution remains enclosed inside [39]. At a fine level, it helps avoid spaghetti code [42], and at a coarser level, it structures the implementation [43] into modules, or layers.

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 [116, 138, 85]. 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 [89], the Hollywood Principle [136], and Monads [146]. Higher-order programming help loosen coupling, thus improve productivity [70].

Closures In languages allowing mutable state, lambda expressions are implemented as closure, to preserve the lexical scope [135]. 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 [144]. 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.

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

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, its platform needs to enforce scalability directly in its design.

Scalability relies on the parallelism allowed by the commutativity of operations execution [29]. 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

The independence, and commutativity of an operation depends on its accesses to shared state. If the operation 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, 62]. 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 [64, 63, 111, 61]. 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 efficiency of paltforms are the synchronization available at a fine-level, and the message-passing available at a coarse-level.

- Fine-level Synchronization

  → avoids communication overhead

- Coarse-level Message-passing

  → allows parallelism

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 support of the community, and the industrial need.

- Community Support

  → grows an ecosystem

- Industrial Need

  → gives a goal for this ecosystem to grow

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.

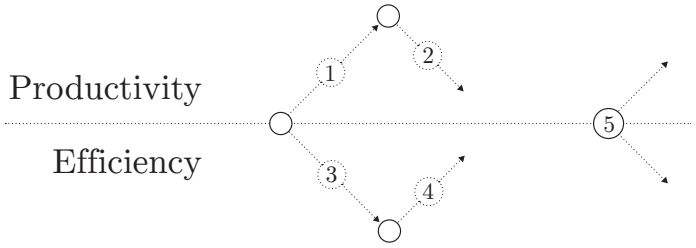


Figure 3.1: Balance between Efficiency and Productivity

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 [130].

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 **??** 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.

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

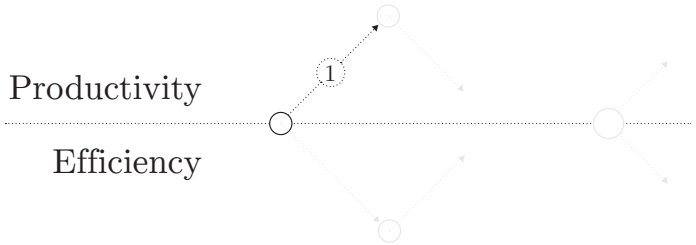


Figure 3.2: Focus on Productivity

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

The very first Object-Oriented Programming (OOP) language was Smalltalk [56]. It defined the core concepts as message passing and encapsulation [3]. Nowadays, the emblematic figures in the software industry are C++ C++ [132] and Java [58]. 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 [123], Miranda [141], Haskell [83] and Standard ML [108]. They provide encapsulation, higher-order programming and lazy evaluation.

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 lan-

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

guages, 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 [84, 142]. 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

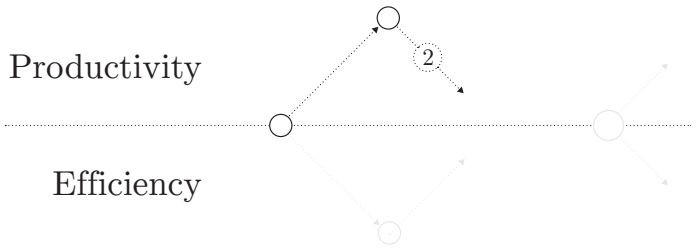


Figure 3.3: Steering back toward Performence Efficiency

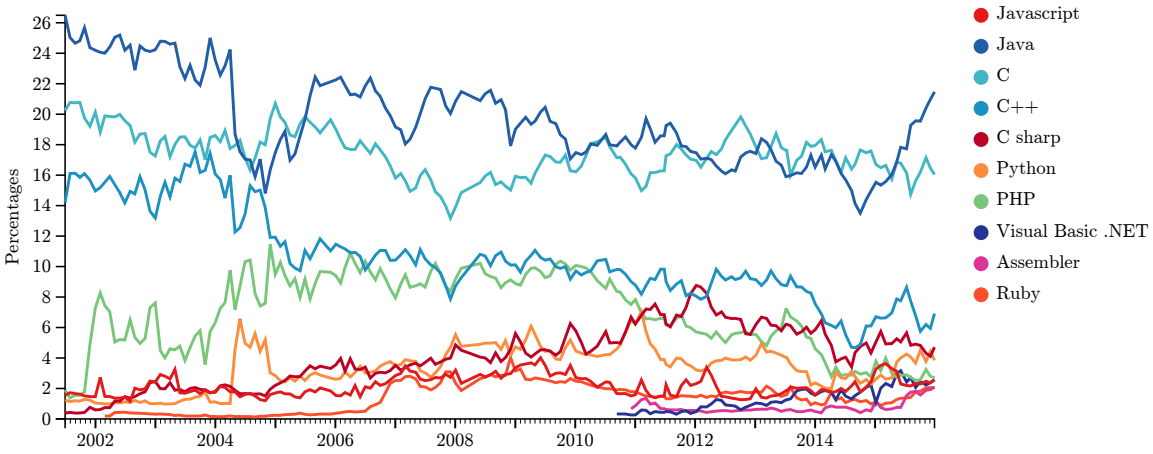


Figure 3.4: TIOBE ranking

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 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

[4] `themostimportantcollaborativedevelopmentplatformgatheringabout9millionsusers.`
[5] `themostimportantQ&Aplatformfordevelopers.`

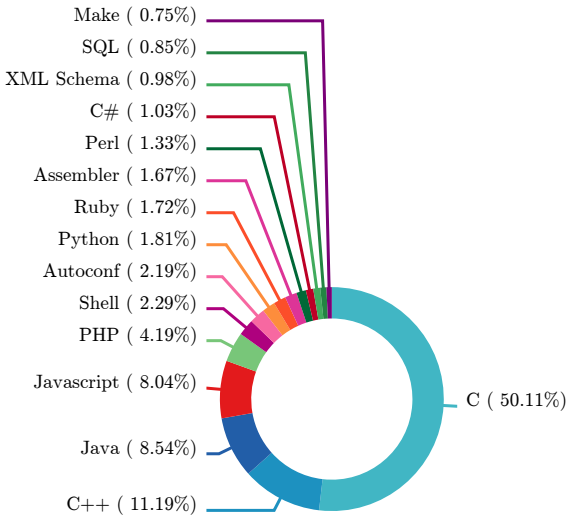


Figure 3.5: Blackduck analysis total

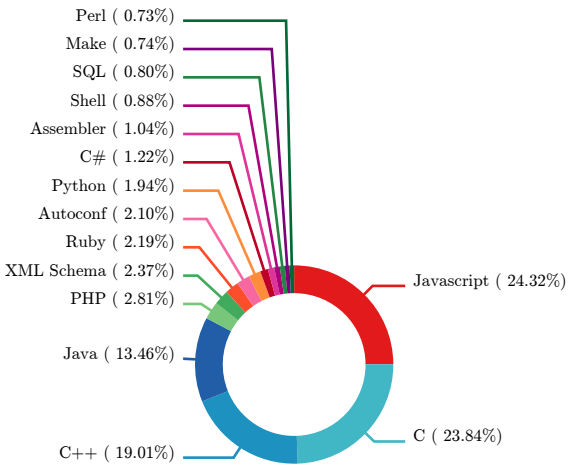


Figure 3.6: Blackduck analysis for 2015

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.

*

*

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

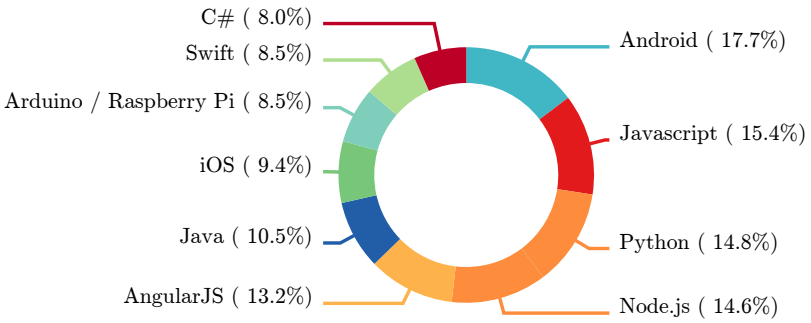


Figure 3.7: Most Wanted Technologies in 2015

TODO include so survey graph

TODO graphical ranking of the tags in StackOverflow

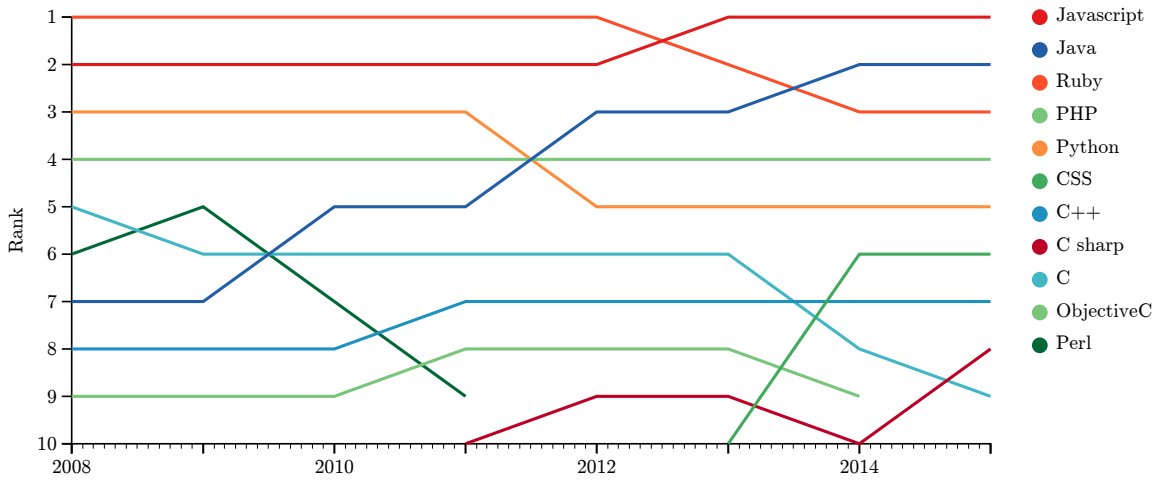


Figure 3.8: Languages Ranks from number of Github projects

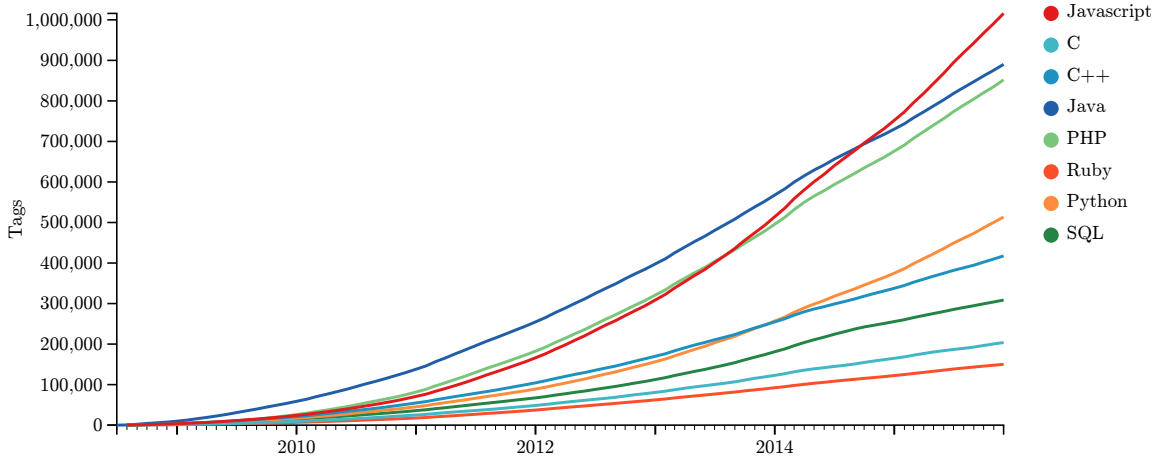


Figure 3.9: StackOverflow Tags evolution

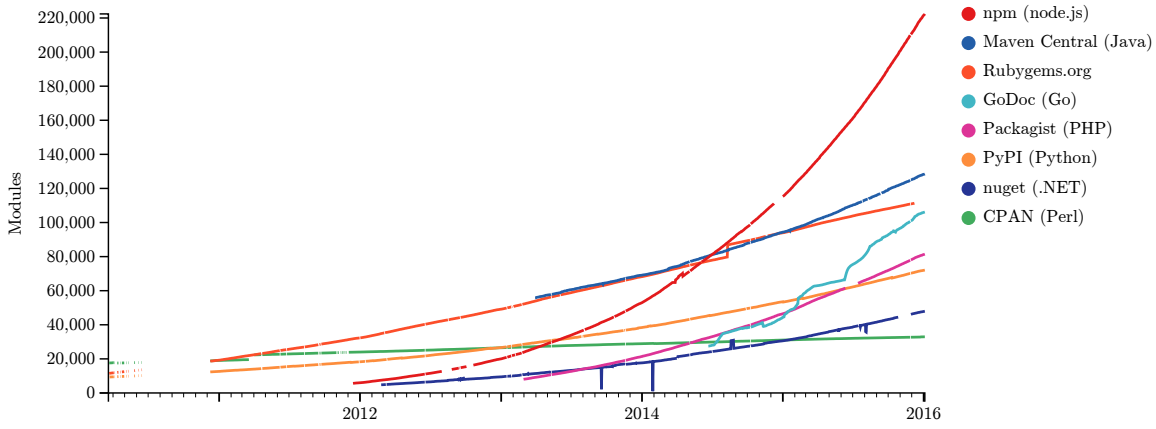


Figure 3.10: Module Counts per package manager

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

*

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.

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

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

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

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 global memory abstraction on which to rely to assure encapsulation and composition – either mutable state or immutable state. 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 [115, 94].

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	4	1		1
Object-Oriented Programming	4	1		1
Functional Programming	1	4		1
Multi Paradigm	4	1		1

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	1
Object-Oriented Programming	5	4	1
Functional Programming	5	1	1
Multi Paradigm	5	4	1

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

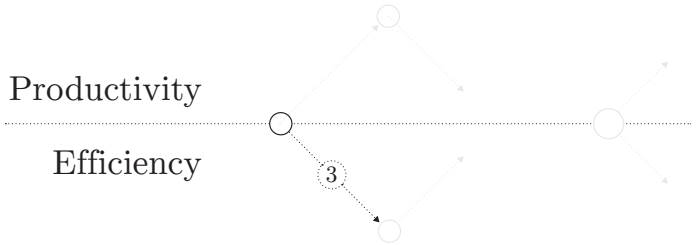


Figure 3.11: Focus on Efficiency

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

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 Hybrid models.

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. And they communicates asynchronously to avoid waiting. A task yields execution to a future tasks to complete the communication. An help with asynchronous programming, promise were introduced [101]. A promise is a placeholder available immediately and allowing to defer operations for when the expected result is available. Promises forms chains of operations similarly to a pipeline.

This execution model 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 [94], Node.js[10] and Vert.X[11]. As well as some web servers like Flash [115], Ninja [59] thttpd[12] and Nginx[13].

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

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

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 [96, 74, 72, 73, 8]. They are based on instructions provided by transactional memories [68] that combine read and write instructions, They provide concurrent implementations of basic data-structures such as linked list [143, 140], queue [134, 150], tree [119] and stack [71].

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 [43], and scheduled in parallel with fork/join instructions [120, 52, 98]. 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 [40], guarded commands [41], guarded region [67] and monitors [80].

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].

Hybrid Models Hybrid models join the advantage of sequential waiting from thread, with the advantage of cooperative scheduling from events-driven models. The implementations of hybrid models are libasync [34], InContext [**Yoo2011**], Fibers [3], Capriccio [17], Monadic hybrid concurrency [100] and Scala Actors [66]. For example, cooperative threads, or fibers, avoid 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[14].

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

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 [95, 122]. But because of the lack of memory isolation, the concurrent tasks are not scheduled in parallel.

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
Hybrid Models	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 [48] 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) [11, 36, 37] and Multiple Program Multiple Data (MPMD) [26, 24]. SPMD defines a single program replicated on many processing units [33, 88, 25] – it is derived from the multi-threading programming model presented in section **??**. While MPMD defines multiple parallel tasks in the implementation [60, 50, 49].

*

⚠ schema of SPMD and MPMD

This section presents MPMD platforms allowing to define isolated tasks. It presents theoretical and programming models on asynchronous communication and 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 [95, 122]. The Actor model allows to express the causal ordering of computation as a set of parallel actors communicating by asynchronous messages [75, 76, 30]. 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 [97]. The Actor model takes these physical limitations in account [77].

Similarly, coroutines are autonomous programs which communicate with adjacent modules as if they were input and output subroutines [32]. It defines a pipeline to implement multi-pass algorithms. Similar works include the Communicating Sequential Processes (CSP) [79, 20], and the Kahn Networks [92, 93].

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
Hybrid Models	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

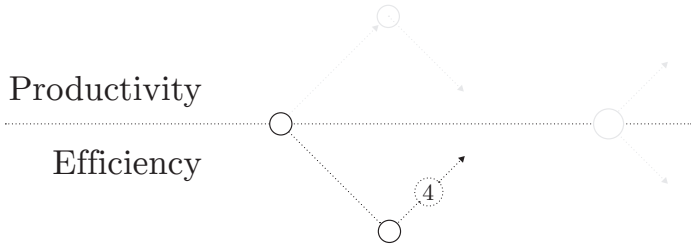


Figure 3.12: Steering back toward Productivity

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
Hybrid Models	0	1		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 [10, 110, 9], Scala [114], Akka[15] and Play[16]. 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 [10, 110, 9]

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 [31, 38, 107, 57]. 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 [38].

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[17], EJB[18] and Spring[19]. It allows to adjust the granularity of tasks to help developers to better fit the tasks organization with the modular organization [2].

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

More recently, Microservices are tackling the same challenge on the web [46, 51, 109]. Some examples of Microservices are Seneca[20]. 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 [86, 154], Apache Hive [139], Timestream [117], Shark [152].

Pipeline Architecture The pipeline architecture introduced by SEDA [149] 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 [59] and Flash [115] 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.

[20] `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
Hybrid Models	0	1		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 pro-

duce 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
Hybrid Models	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
Hybrid Models	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 3.4.1.1 presents Compilers, and section 3.4.1.2

presents Runtimes.

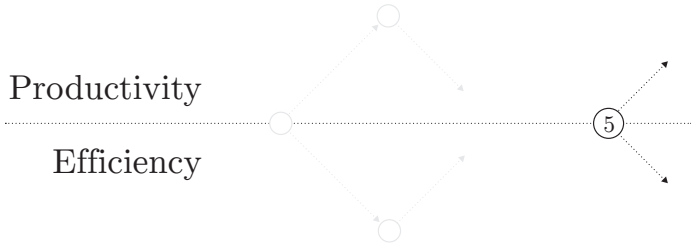


Figure 3.13: Focus on Adoption

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 [16]

.

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 [116]. This section presents the state of the art to extract parallelization from sequential programs through code transformation and compilation.

*

read and include [23]

Parallelism Extraction Extracting parallelism from a sequential implementation is a hard problem [90]. A compiler needs to identify the commutative operations to parallelize their executions [124, 29].

An important work was done to parallelize loop iterations [106, 5, 27, 12, 118], particularly using the polyhedral compilation method [14]. Examples of polyhedral compilers are . To improve performance gains outside of loops, some compilers identify the data-flow parallelism on the whole program [15, 23, 99]. Moreover, the data-flow representation and execution of a program is well suited for modern data processing applications [45], as well as web services [125].

Mutable closures required for higher-order programming remains a challenge to parallelize because of the memory references shared across the program [69, 112, 105]. 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, 87, 129, 148] which allows to infer commutativity. However, this analysis is not sufficient to track the dynamic control-flow of higher-order functions [127] 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 [102, 128, 54, 65, 121, 53, 18]. It is successfully used for security applications [82, 91, 153, 103, 28, 44]*.

⚠ 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 [145, 45]. Such annotations are especially relevant for accelerators such as GPUs or FPGAs, because the development effort yields huge performance improvements [137]. Examples of such compilers are OpenMP [35], OpenCL [131], CUDA [113], Cg [104], Brook [21] and Liquid Metal [81].

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 [125, 126]. 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	②	②		②
Dynamic Distribution	②	②		②
Polyhedral Compiler	②	②		②
Annotation Compiler	②	②		②

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 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. It establish that the platforms favoring one eventually sacrifice the other. The adoption, and usage of these platforms prove that none of these compromises are sustainable.

Section 3.1.1 highlighted that productivity requires modularity through encapsulation and composition. The latter requires higher-order programming which relies on a global memory abstraction as explained in section 3.2.3. On the other hand, section 3.1.2 explains that efficiency requires a balance between fine-grain level shared state with synchronization and coarse-grain level independence with message-passing. And section 3.3.3 explains that this discontinuity between fine-grain level and coarse-grain level avoids productivity. The absence of a global memory abstraction reserves efficient platforms for an elite of developers. No platform can support simultaneously productivity and efficiency.

Moreover, section 3.1.3 explains that for a platforms to be sustainable, it needs to be adopted both by the industry and the community. The industry requires efficiency, while the requires productivity. The previous chapter concludes that no platform is able to follow a project from its needs in productivity in the early beginning to its needs in efficiency during the maturation of the project. Indeed, no platform can provide either productivity and efficiency. All the platforms tends to be stucked in a compromise between these two goals, and cannot follow the evolution required for this compromise. They make the most of this compromise only at a certain point in the evolution of the application. They either become useless, or are too complicated to begin with.

*

⚠ TODO dependency schema of these highlights

Discontinuous 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, a platform meets the requirements of the project only temporarily. 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.

To avoid these consequences, platforms would need to support productivity to allow the community to experiment, and organically start projects. And then con-

tinuously shift toward efficiency as the project evolves, and requires it. The next chapter presents a solution oriented toward that goal.

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	1
Object-Oriented Programming	5	4	1
Functional Programming	5	1	1
Multi Paradigm	5	4	1
Event-driven programming	5	5	3
Lock-free Data-Structures	5	1	3
Multi-threading programming	4	5	3
Hybrid Models	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

Seamless Shift From Productivity to Efficiency

Contents

The evolution of the economical constraints of a web application requires to continuously shift from productivity to efficiency. The incompatibility between the two organizations implies technological ruptures during this evolution. Huge developing efforts are pulled to translate manually from one organization into the other, and later to maintain the implementation despites its unmaintainable nature.

The proposition developed in this thesis is introduced in section 4.1, and then developed throughout this chapter.

4.1 Proposition

This thesis proposes a platform allowing a seamless shift of focus to follow the development of a web application from the productivity required in the early beginning until the efficiency required during maturation. The proposed platform allows to develop applications targeting an event-driven platform allowing productivity, and transforms them so as to execute them on a pipeline architecture allowing efficiency.

Node.js is an efficient event-driven execution model to implement a web application. Javascript features higher-order programming, dynamic typing and a global memory abstraction. Because of these features, it is very productive. However, the efficiency of this execution model is limited by the sequentiality of execution required to preserve exclusivity of memory accesses.

On the other hand, the pipeline execution model doesn't present the same limitation. It enforces memory isolation between stages allowing the parallel execution required for efficiency. But this isolation limits the productivity of this execution model.

The difference in the memory abstractions between the two execution models is illustrated in figure 4.1.

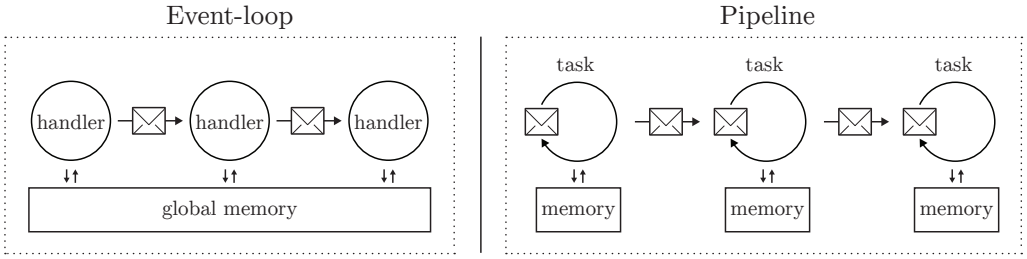


Figure 4.1: Differences of memory abstraction

Despite this difference, these two execution models present interesting similarities. They both organize the execution as a sequence of tasks causaly scheduled.

This thesis proposes an equivalence between the event-driven execution model and the pipeline execution model. It distributes the global memory of the former into memory isolated stages of the latter. It transforms an event-driven application to run on a pipeline architecture.

4.1.1 Continuous Development

This transformation allows a continuity of compromises between productivity and efficiency to continuously follow the shift of focus during development. Developers keep two organizations of the implementation of an application. The productive organization is based on the event-driven execution model. It helps to maintain the application. The efficient organization is the transformed application targeting the pipeline execution model.

At first, the focus remains on the productivity of development rather than the efficiency of execution. The development begins with the event-driven model to take advantage of the productivity of the global memory abstraction. The execution resulting from the transformation is as efficient as the original event-driven execution model.

During the maturation of the application, the focus continuously shift towards efficiency. The transformation distribute the global memory into isolated stages as much as possible. It allows developer to identify the dependencies in this global memory avoiding the distribution. They can identify these dependencies, and arrange the implementation accordingly to allow parallelism. It helps developers to enforce efficiency through continuous iteration, instead of disruptive shifts of technology.

4.1.2 Equivalence

The next paragraphs introduces the equivalence between the the event-driven execution model and the pipeline execution model. The equivalence is broken down in two steps. The first step identifies the rupture points in the control flow separating the stages of the pipeline. The second step enforces isolation of memory between these stages, and replaces synchronization with message passing to preserve the invariance.

4.1.2.1 Rupture Point

In the pipeline architecutre, each stage has its own thread of execution independent from the others. Whereas the handlers in the event-driven execution model are executed sequentially. Despite this difference, the execution of a handler is as independent as the execution of a stage of a pipeline. The call stacks of two handlers are isolated, as illsutrated in figure 4.2. Indeed, a handler holds the execution until all synchronous function calls terminates. The asynchronous function call - the callee - between the caller and its continuation represents a rupture between the two call stacks. And the call stack of the continuation is independent from the call stack of the caller and the callee. This asynchronous callee represents a rupture point between two handlers. It is equivalent to a data stream between two stages in the pipeline architecture. It sends a message between the callee and the continuation.

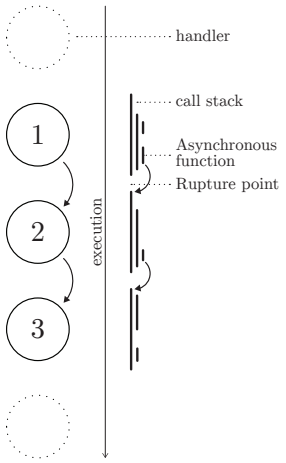


Figure 4.2: Rupture point

The detection of rupture points allows to map a pipeline architecture onto the implementation following the event-loop model. The proposed platform detects rupture points defining stages. This detection is fully addressed in the next chapter, in sections 5.1.3 and 5.2.1. It presents the extraction of a pipeline of concurent tasks from a Javascript application. However, these stages still require a global memory. They can't be executed in parallel.

4.1.2.2 Invariance

A global memory requires the sequential execution of handlers which implies the total scheduling with a queue, as illustrated in figure 4.3. Whereas message passing only requires causal scheduling of handlers which allows parallelism. Yet, the causal scheduling of tasks is sufficient to assure the correctness of the execution. If the handlers didn't rely on the global memory, they could be executed in parallel, as long as their causalities are respected, as illustrated in figure 4.4.

If two handlers causally related rely on the same memory region, the global memory can be replaced by sending the updated memory by message. As illustrated in figure 4.5, each handler has access only to its own memory. The upstream handler communicate the memory update to the downstream handler. However, if the downstream handler modifies this memory, it is not possible to distribute the memory.

If two handlers not causally related rely on the same memory region, they can

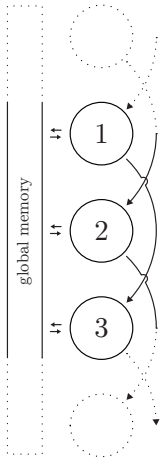


Figure 4.3: Total scheduling

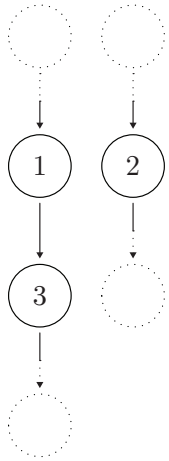


Figure 4.4: Causal scheduling

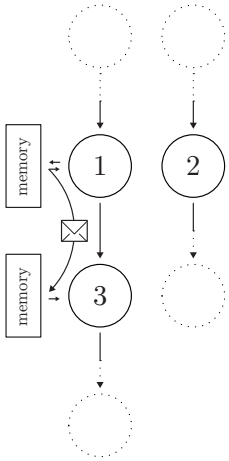


Figure 4.5: Message passing memory update

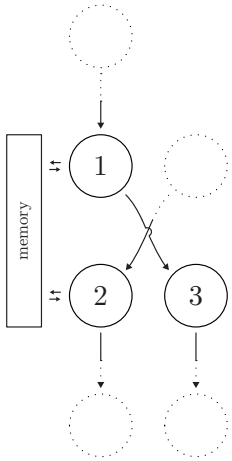


Figure 4.6: Sequential execution

access it in any order. They need to be scheduled sequentially to maintain the exclusivity of access, as illustrated in figure 4.6. The distribution of the global memory is fully addressed in section 5.2.2.

By distributing the global memory following these rules, the sequential scheduling can be loosen to causal scheduling to some extent, while preserving correctness. This distribution only depends on the memory dependencies between handlers. Developers can continuously iterate on implementation to loosen the depenencies between handlers to improve efficiency.

4.1.2.3 Transformation

Figures 4.7 and 4.8 illustrate the two steps of the transformation in the context of the execution models. Figure 4.7 shows the identification of each handler from the event-driven execution model into a stage of the pipeline execution model. Figure 4.8 shows the distribution of the global memory of the event-driven execution model into the different stages of the pipeline execution model.

The next section presents these two execution models and focus on the important details for this transformation.

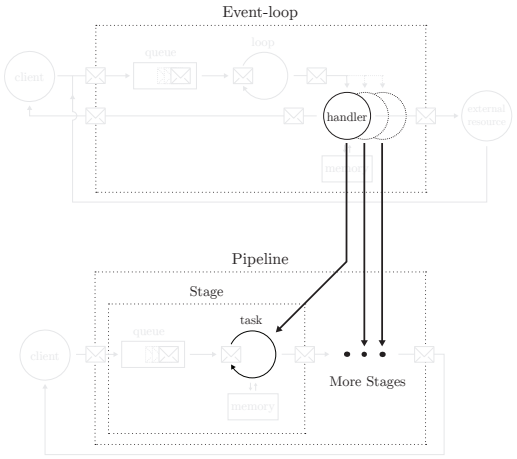


Figure 4.7: Equivalence between handlers and tasks

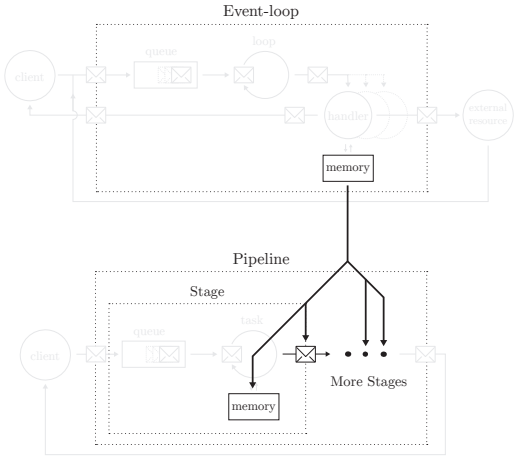


Figure 4.8: Distribution of the global memory abstraction with message passing

4.2 Execution Models

4.2.1 Event-Driven Execution Model

The event-driven execution model processes a queue of asynchronous events by cooperatively scheduling handlers. To respond to an event, the associated handler can directly respond to the source of the event. Or it can request an external resource, and chain another handler to later process the initial event with the resource response, as illustred in figure 4.9. The developer defines each handler as a continuation and defines their causality using the continuation passing style [147, 70].

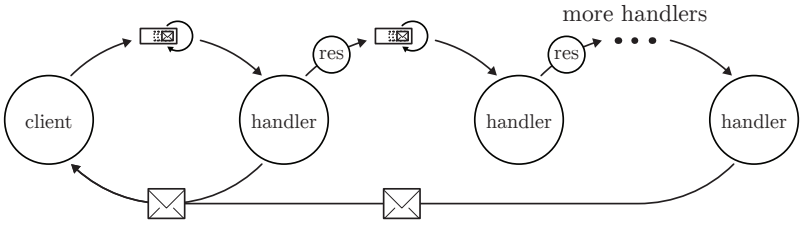


Figure 4.9: Chain of continuations

4.2.1.1 Continuation Passing Style

A continuation is a function passed as an argument to a function call. The continuation is invoked later, at the completion of the callee, to continue the execution. When the callee is asynchronous, it allows not to block the caller until its completion. At its invocation, the continuation retrieves both the caller context, through a closure, and the result. Listing 4.1 illustrates an example of continuation in *Node.js*.

```
callee(input, function continuation(error, result) {
  if (error)
    throw error;

  console.log(result);
});
```

Listing 4.1: Example of a continuation

The continuation passing style lacks the chained composition of multiple asynchronous operations. Promises improve the composition of continuation. They allow to arrange such a sequence of asynchronous operations in a chain, similar to a pipeline.

4.2.1.2 Promise

In the asynchronous paradigm, the control over the asynchronous execution flow is defined with continuations. A Promise is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. In Javascript, promises expose a `then` method which expects a continuation to continue with the result.

The listing 4.2 illustrates this chained composition. The functions `callee_promise_2` and `callee_promise_3` return promises when they are executed. They are executed asynchronously, so these promises are not available synchronously. The method `then` synchronously returns intermediary Promises to bridge with the asynchronous promises. The former resolve when the latter resolve. This behavior allows to arrange the continuations as a flat chain of calls, instead of an imbrication of continuations.

```
callee_promise_1(input)
.then(callee_promise_2, onError)
.then(callee_promise_3, onError)
.then(console.log, onError);

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

Listing 4.2: Example of a chain of Promises

Promises allow to easily arrange the execution flow in parallel or in sequence according to the required causality. Programmers are encouraged to arrange the

computation as series of steps to process incoming events and yield outcoming events. In this sense, Promises are an intermediate step toward the pipeline execution model.

4.2.2 Fluxional execution model

This section presents an execution model inspired by the pipeline architecture. It is the targeet for the transformation presented in this thesis. It intends to provide scalability to web applications with a granularity of parallelism at the function level.

Functions are encapsulated in autonomous execution containers with their state, so as to be mobile and parallel, similarly to the actors model. And the communications are similar to the dataflow programming model, which allows to reason on the throughput of these streams, and to react to load increases [13].

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

Figure 4.10: Syntax of a high-level language to represent a program in the fluxional form

The fluxional execution model executes programs written in the fluxional language, whose grammar is presented in figure 4.10. An application ⟨program⟩ is partitioned into parts encapsulated in autonomous execution containers named *fluxions* ⟨flx⟩. The following paragraphs present the *fluxions* and the messaging system to carry the communications between *fluxions*.

A *fluxion* ⟨flx⟩ is named by a unique identifier ⟨id⟩ to receive messages, and might be part of one or more groups indicated by tags ⟨tags⟩. A *fluxion* is composed of a

processing function ⟨fn⟩, and a local memory called a *context* ⟨ctx⟩.

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

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

There are two types of streams, *start* and *post*, which correspond to the nature of the rupture point producing the stream. A *start* rupture point starts a chain of continuations, while a *post* rupture point is a continuation in a chain. The variables defined before the *start* are available for the whole chain, and require synchronization for concurrent execution of the same chain. Whereas the varaibles defined inside the chain are available only in one chain, and doesn't require synchronization. The two types and implications of rupture points are further detailed in section 5.2.1. *Start* rupture points are indicated with a double arrow (↠ or `>>`) and *post* rupture points with a simple arrow (→ or `->`).

The fluxional execution model is illustrated through an example transformation in the next section.

4.2.3 Example

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      res.send(err || template(count, data));
9    });
10 });
11
12 app.listen(8080);
```

Listing 4.3: Example web application

The example application in listing 4.3 reads a file, and sends it back along with a request counter. The `handler` function, line 5 to 10, receives the input stream of requests. The `count` variable at line 3 counts the requests, and needs to be saved between two messages receptions. The `template` function formats the output stream to be sent back to the client. The `app.get` and `res.send` functions, lines 5 and

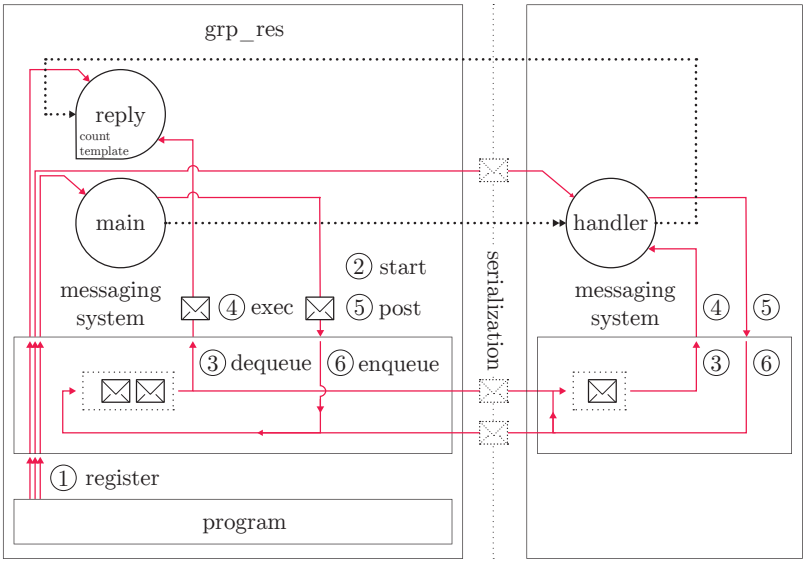


Figure 4.11: The fluxional execution model in details

8, interface the application with the clients. Between these two interface functions is a chain of three functions to process the client requests : `app.get` $\twoheadrightarrow$ `handler` $\rightarrow$ `reply`. This chain of functions is transformed into a pipeline, expressed in the high-level fluxional language in listing 4.4. The transformation process between the source and the fluxional code is explained in 5, in section 5.2.1.

The execution is illustrated in figure 4.11. The dashed arrows between fluxions represent the message streams as seen in the fluxional application. The plain arrows represent the operations of the messaging system during the execution. These steps are indicated by numeroted circles. The *program* registers its fluxions in the messageing system, ①. The fluxion *reply* has a context containing the variable `count` and `template`. When the application receives a request, the first fluxion in the stream, *main*, queues a `start` message containing the request, ②. This first message is to be received by the next fluxion *handler*, ③, and triggers its execution, ④. The fluxion *handler* sends back a message, ⑤, to be enqueued, ⑥. The system loops through steps ③ through ⑥ until the queue is empty. This cycle starts again for each new incoming request causing another `start` message.

```
1 flx main & grp_res
2 >> handler [res]
3   var app = require('express')(),
4       fs = require('fs'),
5       count = 0;
6
7   app.get('/', >> handler); //
```

```
8   app.listen(8080);
9
10  flx handler
11  -> reply [res]
12    function handler(req, res) {
13      fs.readFile(__filename, -> reply); //
14    }
15
16  flx reply & grp_res {count, template}
17  -> null
18    function reply(error, data) {
19      count += 1; //
20      res.send(err || template(count, data)); //
21    }
```

Listing 4.4: Example application expressed in the high-level fluxional language

The chain of functions from listing 4.3 is expressed in the fluxional language in listing 4.4. The fluxion `handler` doesn't have any dependencies, so it can be executed in a parallel event-loop. The fluxions `main` and `reply` belong to the group `grp_res`, indicating their dependency over the variable `res`. The group name is chosen arbitrarily. All the fluxions inside a group are executed sequentially on the same event-loop, to protect the shared variables against concurrent accesses.

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

This execution model allows to parallelize the execution of an application as a pipeline, as with the fluxion `handler`. And some parts are replicated, as could be the group `grp_res`. This parallelization improves the efficiency 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.

Yet, the parallelization is limited by the dependencies between fluxions. A developer can ignore these dependencies at first, to focus on productivity. And then continuously tune the implementation to remove these dependencies and improve efficiency. This continuous tuning avoid the disruptive shifts of technology required by current platforms.

Chapter 5

Implementation

Contents

The two step of the transformation are the identification of a pipeline, and the isolation of its stages. This chapter presents the technical implementations of these two steps in the transformation from the event-driven execution model to the pipeline architecture As presented in figure 5.1, the transformation described in the previous chapter was implemented incrementally in two compilers.

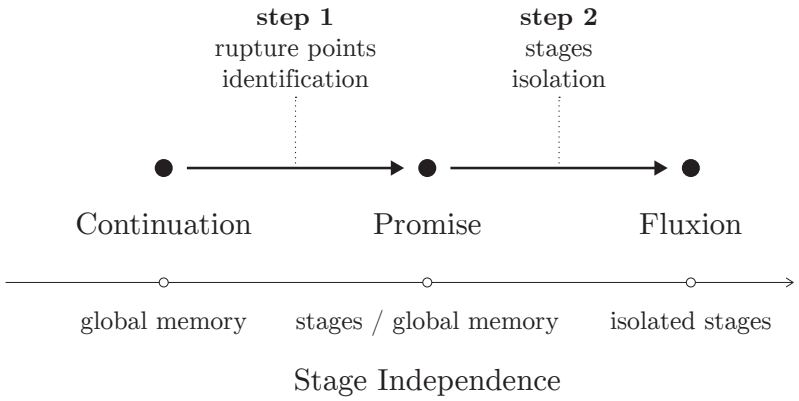


Figure 5.1: Roadmap

The first compiler focus on the identification of simple chains of causality between continuations to layout these chains as Promises. Promises bring more control over the asynchronous flow than the chaining of causal sequentiality. They impose a different convention than continuations on how to hand back the outcome and errors of the deferred computation. This difference brings unnecessary complexity to the transformation. To rule out this difference between continuations and Promises, section 5.1 introduces a simpler specification to Promise, called Due.

The second compiler detects all the chains of causality between continuations and encapsulate them in fluxions. It isolates the fluxions when possible to allow the parallelism required for efficiency. The second compilers is introduced in section 5.2.

5.1 Dues

5.1.1 Due

A Due is an object used as placeholder for the eventual outcome of a deferred operation. They are essentially similar to ECMAScript Promises[1], except for the con-

[1] `http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects`

vention to hand back outcomes. They use the *error-first* convention, like *Node.js*, as illustrated line 5 in listing 5.1. The implementation of Dues and its tests are available online[2].

5.1.1.1 Usage

```
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 5.1: Example of a due

In listing 5.1, the function `my_fn_due` synchronously returns a due as a placeholder for its outcome. The `then` method of the due allows to define a continuation to continue the execution after retrieving the outcome, like line 5. 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.

5.1.1.2 Creation

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

Listing 5.2: Creation of a due

In listing 5.1, line 1, the `mock` method wraps the original function `my_fn` in a Due-compatible function `mocked_fn`. The `mock` method is detailed in listing 5.2 to illustrate the creation of a Due. It returns a Due compatible function, `mocked_fn`, line 2. That is a function that returns a Due, instead of expecting a continuation.

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

At the execution of `mocked_fn` the Due to be returned is created line 6, with the original function passed as argument. The original function `my_fn` is executed during the creation of the Due. The `settle` function provided is passed as a continuation line 7 for the original function to settle the returned Due. When the original function completes, it calls `settle` to settle the Due and save the outcome. This outcome can then be retrieved with the continuation provided by the `then` method.

5.1.1.3 Composition

Dues arrange the execution flow as a chain of actions to carry on inputs. The composition of Dues in a chain is illsutrated in listing 5.3.

```
1  var Due = require('due');
2
3  var my_fn_due_1 = Due.mock(my_fn_1),
4      my_fn_due_2 = Due.mock(my_fn_2),
5      my_fn_due_3 = Due.mock(my_fn_3);
6
7  my_fn_due_1(input)
8  .then(my_fn_due_2)
9  .then(my_fn_due_3)
10 .then(console.log);
```

Listing 5.3: Dues are chained like Promises

The `then` method of the current Due returns an intermediary Due that settles when the Due returned by the passed continuation settles. As example, in listing 5.3 the Due returned by the `then` method line 8 settles when the Due returned by `my_fn_due_2` settles. It allows to chain continuations one after the other, instead of the nested composition of continuations.

5.1.2 From Continuations to Dues

The equivalence between continuations and Dues allows the transformation of a nested imbrication of continuations into a chain of Dues. To preserve the semantic, this transformation imposes limitations on the execution order, the execution linearity and the scopes of the variables used in the operations.

5.1.2.1 Execution order

The transformation of a simple continuation is illsutrated in figure 5.1.2.1 The compiler spots function calls similar to the abstraction (5.1). It wraps the function fn into the function $fn_{\mathbf{due}}$ to return a Due, as presented in section 5.1.1.2 And it relocates the continuation in a call to the method **then**. The result is similar to the abstraction (5.2). The differences are highlighted in bold font.

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

$$\downarrow$$

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

Figure 5.2: Simple transformation

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 function fn must be asynchronous to preserve the execution order.

5.1.2.2 Execution linearity

The transformation of a chain of continuations into a chain of Dues is illustrated in figure 5.1.2.2. The compiler transforms a nested imbrication of continuations similar to the abstraction (5.3) into a flatten chain of calls encapsulating them, as abstraction (5.4).

Because of the repetition, a call inside a loop yields multiple Dues to chain, while only one is returned to continue the chain. Loops modify the linearity of the execution flow. To preserve the semantic, a chain of Dues must not contain any loop. Similarly, a function definition outside a function call breaks the execution linearity. This defined function might not be executed synchronously, in the current chain of Dues. It prevents the nested call to return the Due expected to continue the chain. Therefore, the compiler breaks the chain of Dues when loops or function definitions are encountered.

On the other hand, conditional branching leaves the execution linearity and the semantic intact. If the nested asynchronous function is not called due to branching, the execution of the chain stops as expected, either with continuations or Dues.

$$
\begin{array}{l}
fn1([arguments], cont1\{ \\
\quad declare\ variable \leftarrow result \\
\quad fn2([arguments], cont2\{ \\
\quad\quad print\ variable \\
\quad \}) \\
\}) \qquad (5.3)
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\textbf{declare variable} \\
fn1_{\textbf{due}}([arguments]) \\
\textbf{.then}(cont1\{ \\
\quad variable \leftarrow result \\
\quad \textbf{return}\, fn2_{\textbf{due}}([arguments]) \\
\}) \\
\textbf{.then}(cont2\{ \\
\quad print\ variable \\
\}) \qquad (5.4)
\end{array}
$$

Figure 5.3: Composition transformation

5.1.2.3 Variable scope

In abstraction (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 abstraction (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. Most imperative languages like C/C++, Python, Ruby or Java present a lexical scope, which defines variables scopes statically. In Javascript, however, the statements `with` and `eval` modify the scope dynamically. The compiler excludes programs using these statements.

5.1.3 Due Compiler

The Due compiler automates the application of this equivalence on existing Javascript projects. The compilation process is made of two important steps, the identification of the continuations, and the generation of chains. The compiler is available as a standalone web application to reproduce the tests [3].

5.1.3.1 Identification of continuations

The first compilation step is to identify the continuations and their imbrications. The compiler transforms only *in situ* continuations. Modifiying continuations that

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

are named functions, and defined elsewhere impacts the semantic.

To detect continuations, the compiler looks for callbacks. That is a function definition within the arguments of a function call. Not all detected callbacks are continuations, whereas the equivalence is applicable only on the latter. A continuation is a callback invoked only once, asynchronously. 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 holds only on semantical differences.

To recognize the two, 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. For example, the *node.js* method `fs.readFile` is asynchronous. Its argument is a continuation. But it can be overridden by developers to be synchronous, or to execute the callback several times. The compiler leaves the identification of compatible continuations among the identified callbacks to the developer.

5.1.3.2 Generation of chains

Continuations structure the execution flow as a tree, whereas a chain of Dues arranges it sequentially. A parent continuation can execute several children, while a Due allows to chain only one. The second compilation step is to identify the trees 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. A legitimate child is a unique continuation which contains another continuation to chain. If there are several continuations that continue the chain, none are the legitimate child. And the non legitimate children start new chains of Dues. In figure 5.1.3.2, the continuation `cont2` is a legitimate child.

This step transforms each tree of continuations into several chains of continuations that translate into sequences of Dues.

This pipeline of Dues represents only a part of the pipeline present in an application. It doesn't account for the initiation of the pipeline which allow the streaming of data. More exactly, in the case of streaming data, Dues are created for each new datum in the stream.

Moreover, the Dues still rely on a shared memory to communicate. They are don't enforce the memory isolation required for parallelism.

```
caller1([args], function cont1(){
  // ① ...
  caller2([args], function cont2(){
    // ③ ...
    caller3([args], function cont3(){
      // ⑤ ...
    });
    // ④ ...
    caller4([args], function cont4(){
      // ⑥ ...
    });
  });
  // ② ...
})
```

Listing 5.4: Nested calls of continuations

→

```
caller1([args])
.then(function cont1(){
  // ① ...
  return caller2([args])
  // ② ...
})
.then(function cont2(){
  // ③ ...
  caller3([args], function cont3(){
    // ⑤ ...
  });
  // ④ ...
  caller4([args], function cont4(){
    // ⑥ ...
  });
})
```

Listing 5.5: Chain of Due

Figure 5.4: Transformation of a tree of continuations into a chain of Due

The next section address these two limitations, with the extraction of a pipeline of fluxions.

The Due compiler is an intermediary step toward the second compiler presented in section 5.2.

5.1.3.3 Evaluation

A set of Javascript projects likely to contain continuations were compiled to validate the compiler. This section presents the results of these tests.

This compiler has been tested over 64 *Node.js* packages from the node package manager (npm[4]). 55 packages were incompatible with the compiler, 9 packages were compiled with success.

The compilation of a project requires user interaction. To conduct the test in a reasonable time, the test set was limited to a minimum of about 50 projects. All the projects in the set were selected from the *Node Package Manager* database to restrict the set to *Node.js* projects. They all depends on the web framework *express*, but not on the most common Promises libraries such as *Q* and *Async*. They use the test frameworks *mocha* in its default configuration. These tests are used to validate the compilation results. The test set finally contains 64 projects. This subset is very small, and cannot represent the wide possibilities of Javascript. However, it is sufficient to represents a majority of common cases.

[4] `https://www.npmjs.com/`

Each project passes its own tests before compilation. During the compilation, the compatible continuations were manually identified among the detected callbacks. The compilation result of each project is then tested again with its unmodified test. The compilation result should pass the tests as well as the original project. 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. These projects might use continuations, but the compiler is unable to detect them. The other projects were rejected by the compiler because they contain `with` or `eval` statements, they use Promises libraries didn't filtered 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. 56 of them were manually identified as 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 asynchronous libraries,

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 **??**.

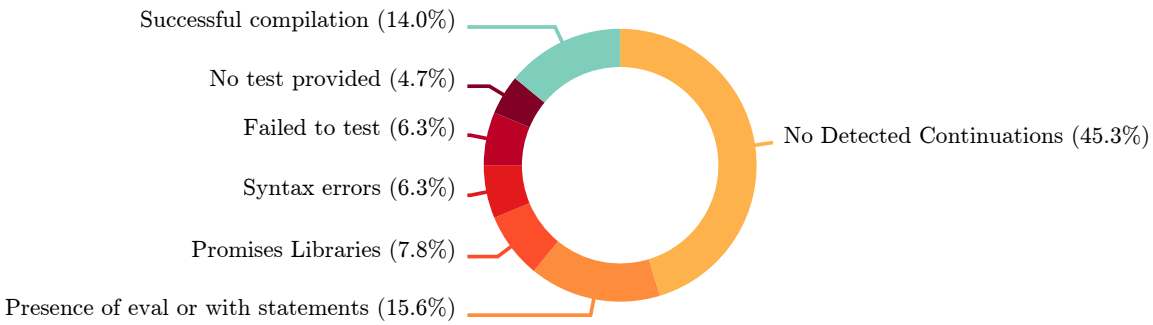


Figure 5.5: Results of the Due compiler evaluation

5.2 Fluxions

The previous section presented a compiler to identify and extract the underlying pipeline in a Javascript application. However, the stages doesn't enforce the isolation required for parallel execution. Moreover, the Dues that constitues the stages of this pipeline

only parts of the pipeline are identified, This section 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.

The identification of the rupture points between fluxions is addressed in section 5.2.1. The isolation between the fluxions, after identification, is addressed in section 5.2.2. Section 5.2.3 presents a real-case test of compilation, and expose the limits of this compiler.

5.2.1 Fluxions Identification

The source languages we focus on should offer higher-order functions and be implemented as an event-loop with a global memory. Javascript is such a language and is often implemented on top of an event-loop, like in *Node.js*. We developed a compiler that transforms a *Node.js* application into a fluxional application compliant with the

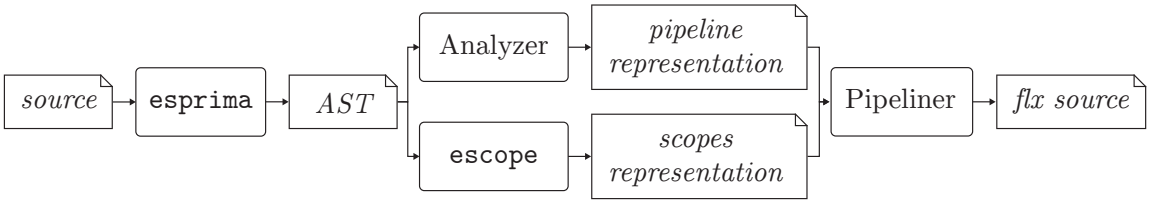


Figure 5.6: Compilation chain

execution model described in section **??**. Our compiler uses the *estools*[5] suite to parse, manipulate and generate source code from Abstract Syntax Tree (AST). And it is tailored for – but not limited to – web applications using *Express*[6], the most used *Node.js* web framework.

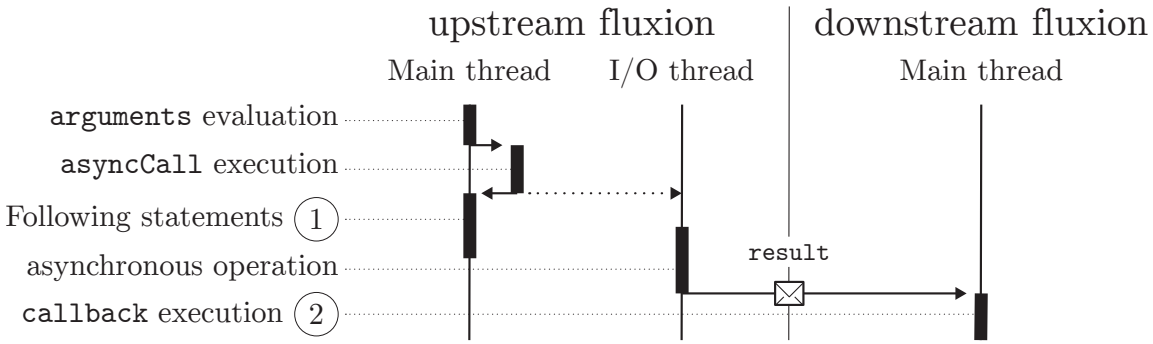
The chain of compilation is described in figure 5.6. The compiler extracts an AST from the source with `esprima`. From this AST, the *Analyzer* step identifies the limits of the different application parts and how they relate to form a pipeline. This first step outputs a pipeline representation of the application. Section **??** 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 **??** explains this second compilation step.

The limit between two application parts is defined by a rupture point. The analyzer identifies these rupture points, and outputs a representation of the application in a pipeline form. Application parts are the stages, and rupture points are the message streams of this pipeline.

5.2.1.1 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 rupture points between two application parts. Figure 5.7 shows a code example of a rupture point with the illustration of the execution of the two application parts isolated into fluxions. The two application parts are the caller of the asynchronous function call on one hand, and the callback provided to the asynchronous function call on the other hand.

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




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

Figure 5.7: Rupture point interface

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with data not available in the caller context. There are three kinds of callbacks, but only two are asynchronous: listeners and continuations. The two corresponding types of rupture points are *start* and *post*.

Start rupture points (listeners) are on the border between the application and the outside, continuously receiving incoming user requests. An example of a start rupture point is in listing 4.3, between the call to `app.get()`, and its listener `handler`. These rupture points indicate the input of a data stream in the program, and the beginning of a chain of fluxions to process this stream.

Post rupture points (continuations) represent a continuity in the execution flow after an asynchronous operation yielding a unique result, such as reading a file, or a database. An example of a post rupture points is in listing 4.3, between the call to `fs.readFile()`, and its continuation `reply`.

5.2.1.2 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 tests if one of the arguments is of type `function`. Some callback functions

are declared *in situ*, and are trivially detected. For variable identifiers, and other expressions, the analyzer tries to detect their type. The analyzer walks back the AST to track their assignations and modifications, so as to determine their last value.

5.2.2 Fluxions Isolation

A rupture point eventually breaks the chain of scopes between the upstream and downstream fluxion. The closure in the downstream fluxion cannot access the scope in the upstream fluxion as expected. The pipeliner step replaces the need for this closure, allowing application parts to rely only on independent memory stores and message passing. It determines the distribution using the scope representation, which represents the variables' dependencies between application parts. Depending on this representation, the compiler can replace the broken closures in three different ways. We present these three alternatives in figure 5.8.

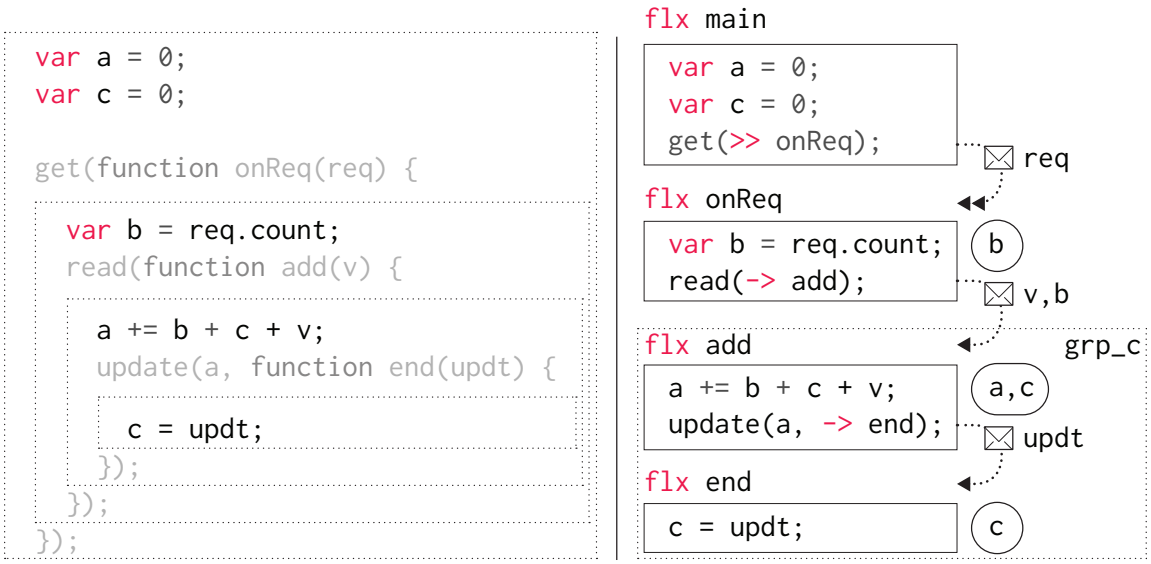


Figure 5.8: Variable management from Javascript to the high-level fluxional language

Scope If a variable is modified inside only one application part in the current *post* chain, then the pipeliner adds it to the context of its fluxion.

In figure 5.8, the variable a is updated in the function add. The pipeliner step stores this variable in the context of the fluxion add.

Stream If a modified variable is read by downstream application parts, then the pipeliner makes the upstream fluxion add this variable to the message stream to be sent to the downstream fluxions. It is impossible to send variables to upstream fluxions, without causing inconsistencies. If the fluxion retro propagates the variable for an upstream fluxion to read, the upstream fluxion might use the old version while the new version is on its way.

In figure 5.8, the variable `b` is set in the function `onReq`, and read in the function `add`. The pipeliner step makes the fluxion `onReq` send the updated variable `b`, in addition to the variable `v`, in the message sent to the fluxion `add`.

Exceptionally, if a variable is defined inside a *post* chain, like `b`, then this variable can be streamed inside this *post* chain without restriction on the order of modification and read. Indeed, the execution of the upstream fluxion for the current *post* chain is assured to end before the execution of the downstream fluxion. Therefore, no reading of the variable by the upstream fluxion happens after the modification by the downstream fluxion.

Share If a variable is needed for modification by several application parts, or is read by an upstream application part, then it needs to be synchronized between the fluxions. To respect the semantics of the source application, we cannot tolerate inconsistencies. Therefore, the pipeliner groups all the fluxions sharing this variable with the same tag. And it adds this variable to the contexts of each fluxions.

In figure 5.8, 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.

5.2.3 Real test case

This section presents a test of the compiler on a real application, gifsockets-server[7]. This test proves the possibility for an application to be compiled into a network of independent parts. It shows the current limitations of this isolation and the modifications needed on the application to circumvent them. This section then presents future works.

```
1 var express = require('express'),
2     app = express(),
3     routes = require('gifsockets-middleware'),
4     getRawBody = require('raw-body');
5
6 function bodyParser(limit) {
```

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

```
7   return function saveBody(req, res, next) {
8     getRawBody(req, {
9       expected: req.headers['content-length'],
10      limit: limit
11    }, function (err, buffer) {
12      req.body = buffer;
13      next();
14    });
15  };
16 }
17
18 app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.writeTextToImages);
19 app.listen(8000);
```

Listing 5.6: Simplified version of gifsockets-server

This application, simplified in listing 5.6, is a real-time chat using gif-based communication channels. It was selected in a previous work [19] from the `npm` registry because it depends on `express`, it is tested, working, and simple enough to illustrate this evaluation. The server transforms the received text into a gif frame, and pushes it back to a never-ending gif to be displayed on the client.

On line 18, the application registers two functions to process the requests received on the url `/image/text`. The closure `saveBody`, line 7, returned by `bodyParser`, line 6, and the method `routes.writeTextToImages` from the external module `gifsockets-middleware`, line 3. The closure `saveBody` calls the asynchronous function `getRawBody` to get the request body. Its callback handles the errors, and calls `next` to continue processing the request with the next function, `routes.writeTextToImages`.

5.2.3.1 Compilation

We compile this application with the compiler detailed in section **??**. Listing 5.7 presents the compilation result. The function call `app.post`, line 18, is a rupture point. However, its callbacks, `bodyParser` and `routes.writeTextToImages` are evaluated as functions only at runtime. For this reason, the compiler ignores this rupture point, to avoid interfering with the evaluation.

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

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

Listing 5.7: Compilation result of gifsockets-server

The compiler detects a rupture point : the function `getRawBody` and its anonymous callback, line 11. It encapsulates this callback in a fluxion named `anonymous_1000`. The callback is replaced with a stream placeholder to send the message stream to this downstream fluxion. The variables `req` and `next` are appended to this message stream, to propagate their value from the `main` fluxion to the `anonymous_1000` fluxion.

When `anonymous_1000` is not isolated from the `main` fluxion, as if they belong to the same group, the compilation result works as expected. The variables used in the fluxion, `req` and `next`, are still shared between the two fluxions. Our goal is to isolate the two fluxions, to be able to safely parallelize their executions.

5.2.3.2 Isolation

In listing 5.7, 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.

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

Closure `next` The function `next` is a closure provided by the `express Router` to continue the execution with the next function to handle the client request. Because it indirectly relies on the variable `req`, it is impossible to isolate its execution with the `anonymous_1000` fluxion. Instead, we modify `express`, so as to be compatible with the fluxional execution model. We explain the modifications below.

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

Listing 5.8: Simplified modification on the compiled result

In listing 5.6, the function `next` is a continuation allowing the anonymous callback, line 11, to call the next function to handle the request. To isolate the anonymous callback into `anonymous_1000`, `next` is replaced by a rupture point. This replacement is illustrated in listing 5.8. The `express Router` registers a fluxion named `express_dispatcher`, line 8, to continue the execution after the fluxion `anonymous_1000`. This fluxion is in the same group `express` as the `main` fluxion, hence it has access to the original variable `req`, and to the original function `next`. The call to the original `next` function is replaced by a placeholder to push the stream to the fluxion `express_dispatcher`, line 5. The fluxion `express_dispatcher` receives the stream from the upstream fluxion `anonymous_1000`, merges back the modification in the variable `req` to propagate the side effects, and finally calls the original function `next` to continue the execution, line 12.

After the modifications detailed above, the server works as expected. The isolated fluxion correctly receives, and returns its serialized messages. The client successfully receives a gif frame containing the text.

Chapter 6

Evaluation

TODO when state of the art ready

Chapter 7

Conclusion

Contents

The web allows a new economic model to emerge, and a tremendous number of business opportunities. To seize these opportunities, a team needs to develop a web application, and grow a business around it. The economical incentives around the technical development changes completely during the growth of this business. In the beginning, the development needs to be productive, to quickly release a product, and iterate with the user feedbacks. While when the project matures, the execution needs to be efficient, to cope with the load of a large user base while limiting the hardware costs.

These two development concerns are incompatible. No platform can provide both performance efficiency, and development productivity at the same time. Moreover, the studied platform propose only a fixed compromised between the two. This work presented a platform to allow the evolution of this compromise to follow the evolution of the economical incentives.

7.1 Summary

The following paragraphs presents the contributions of this thesis.

7.1.1 Fluxional Execution Model

TODO

7.1.2 Pipeline Extraction

TODO

7.1.3 Pipeline Isolation

TODO

7.2 Opening

This thesis brought a possible reconciliation between two concerns in the development of a web application, the efficiency of execution and the productivity of development. However, a third concern is currently increasingly taking importance. The IT industry have an increasing carbon footprint. The impact of this lifestyle on the environment starts only to emerge. It is of crucial importance to limit the carbon

emission of our lifestyle. I leave for future works the reconciliation of the efficient of energy consumption with the two concerns tackled in this thesis.

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. *Programming Erlang*. Pragmatic Programmers, 2007, p. 519. DOI: 10.1017/S0956796809007163.

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

[11] 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.

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

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

[14] 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.

[15] 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.

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

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] Frank Dabek and Nickolai Zeldovich. "Event-driven programming for robust software". In: *Proceedings of the 10th workshop on ACM SIGOPS European workshopn workshop* (July 2002), pp. 186–189. DOI: 10.1145/1133373.1133410.

[35] 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.

[36] 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.

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

[38] 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.

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

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

[41] 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.

[42] 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.

[43] 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.

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

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

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

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

[48] 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.

[49] 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.

[50] 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.

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

[52] 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].

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

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

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

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

[57] 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.

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

[59] 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.

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

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

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

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

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

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

[66] Philipp Haller and Martin Odersky. "Actors That Unify Threads and Events". In: *Coordination 2007, Lncs 4467* (2007), pp. 171–190. DOI: 10.1007/978-3-540-72794-1{_}10.

[67] 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.

[68] 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.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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).

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

[77] Carl Hewitt and Jr Baker Henry. "Actors and Continuous Functionals," in: (Dec. 1977).

[78] 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.

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

[80] 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.

[81] Shan Huang, Amir Hormati, David Bacon, and Rodric Rabbah. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary". In: *ECOOP 2008 – Object-Oriented Programming*. 2008, pp. 76–103. DOI: 10.1007/978-3-540-70592-5{_}5.

[82] 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).

[83] 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.

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

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

[86] 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).

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

[88] 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.

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

[90] 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.

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

[92] 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.

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

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

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

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

[97] 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.

[98] 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.

[99] 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.

[100] Peng Li and Steve Zdancewic. "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives". In: *ACM SIGPLAN Notices* 42.6 (June 2007), p. 189. DOI: 10.1145/1273442.1250756.

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

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

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

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

[105] 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.

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

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

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

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

[110] 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.

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

[112] Jens Nicolay. “Automatic Parallelization of Scheme Programs using Static Analysis”. PhD thesis. 2010.

[113] C Nvidia. “Compute unified device architecture programming guide”. In: (2007).

[114] 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.

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

[116] 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.

[117] 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).

[118] 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).

[119] 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.

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

[121] Veselin Raychev, Martin Vechev, and Manu Sridharan. “Effective Race Detection for Event-driven Programs”. In: *SIGPLAN Not.* 48.10 (Nov. 2013), pp. 151–166. DOI: 10.1145/2544173.2509538.

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

[123] 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.

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

[125] 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.

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

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

[128] GD Smith. "Local reasoning about web programs". In: (2011).

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

[130] 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.

[131] 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.

[132] B Stroustrup. "The C++ programming language". In: (1986).

[133] 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.

[134] 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.

[135] 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.

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

[137] David Tarditi, Sidd Puri, and Jose Oglesby. "Accelerator: using data parallelism to program GPUs for general-purpose uses". In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* 34.5 (Oct. 2006), pp. 325–335. DOI: 10.1145/1168918.1168898.

[138] 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.

[139] 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.

[140] 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.

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

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] M Wand. "Continuation-based multiprocessing". In: *Proceedings of the 1980 ACM conference on LISP and ...* (1980).

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

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

[150] 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.

[151] 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.

[152] 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.

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

[154] 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).