Fluxional compiler : seamless shift from development productivity to performance efficiency, in the case of real-time web applications

Etienne Brodu

February 29, 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

6 CHAPTER 1 INTRODUCTION

1.1 Web development 7
1.2 Performance requirements 7
1.3 Problematic and proposal 8
1.4 Thesis organization 9

10 CHAPTER 2 CONTEXT AND OBJECTIVES

2.1 The Web as a Platform 11
2.1.1 The Language of the Web 11
2.1.2 Highly Concurrent Web Servers 15
2.2 An Economical Problem 17
2.2.1 Disrupted Web Development 18
2.2.2 Seamless Web Development 19

21 CHAPTER 3 SOFTWARE DESIGN, STATE OF THE ART

3.1 Definitions 23
3.1.1 Productivity 23
3.1.2 Efficiency 25
3.1.3 Adoption 26
3.2 Productivity Focused Platforms 27
3.2.1 Modular Programming 28
3.2.2 Steering Back Toward Efficiency 30
3.2.3 Efficiency Limitations 34
3.2.4 Summary 34
3.3 Efficiency Focused Platforms 35
3.3.1 Concurrency 36
3.3.2 Steering Back Toward Producitivity 41
3.3.3 Productivity Limitations 45
3.3.4 Summary 47
3.4 Compromise Between Productivity And Efficiency . . . 49
3.4.1 Abstraction of Tasks Organization 49
3.4.2 Limitation 54
3.4.3 Summary 54

3.5 Discontinuous Developments 55

CHAPTER 4
59 SEAMLESS SHIFT FROM PRODUCTIVITY TO EFFICIENCY

4.1 Proposition 60
4.1.1 Continuous Development 61
4.1.2 Equivalence 61
4.2 Execution Models 65
4.2.1 Event-Driven Execution Model 65
4.2.2 Fluxional Execution Model 67
4.2.3 Examples 68
4.3 Conclusion 72

CHAPTER 5
73 IMPLEMENTATIONS

5.1 Step 1 - Due Compiler 74
5.1.1 Dues 74
5.1.2 From Continuations to Dues 76
5.1.3 Due Compiler 78
5.2 Step 2 - Fluxional Compiler 81
5.2.1 Fluxions Compiler 81
5.2.2 Fluxions Isolation 83
5.2.3 Real test case 84
5.2.4 Limitations 87

CHAPTER 6
89 CONCLUSION

6.1 Summary 90
6.1.1 Models 90
6.1.2 Equivalence 91
6.2 Overall Evaluation 93
6.2.1 Trading Productivity for Efficiency 93
6.2.2 Adoption 94
6.3 Perspectives 94
6.3.1 Just-in-time Compilation 95
6.3.2 Evaluation of the perspective 96
6.3.3 Final Thoughts 97

List of Figures

2.1 Javascript timeline 14
2.2 Event-driven execution model 15
2.3 Pipeline execution model 17
2.4 Comparison of the two memory models 18

3.1 Balance between Efficiency and Productivity 27
3.2 TIOBE ranking 31
3.3 Languages Ranks from number of Github projects 31
3.4 StackOverflow Tags evolution 32
3.5 Module Counts per package manager 32

4.1 Comparison of the two memory models 60
4.2 Rupture point 61
4.3 Sequential scheduling 63
4.4 Causal scheduling 63
4.5 Message passing memory update 64
4.6 Sequential execution 64
4.7 Equivalence between handlers and tasks 65
4.8 Distribution of the global memory abstraction with message passing 65
4.9 Chain of continuations 66
4.10 Syntax of a high-level language to represent a program in the fluxional form 68
4.11 Screenshot from the grumpy console 69
4.12 The fluxional execution model in details 71

5.1 Roadmap 74
5.2 Simple transformation 76
5.3 Composition transformation 77
5.4 Transformation of a tree of continuations into a chain of Due 79
5.5 Results of the Due compiler evaluation 81
5.6 Compilation chain 82
5.7 Rupture point interface 83
5.8 Variable management from Javascript to the high-level fluxional language 83

List of Tables

3.1 Productivity of Modular Programming Platforms 29
3.2 Adoption of Modular Programming Platforms 33
3.3 Efficiency of Modular Programming Platforms 34
3.4 Summary of Modular Programming Platforms 35
3.5 Efficiency of Concurrent and Parallel Programming Platforms 40
3.6 Adoption of Concurrent and Parallel Programming Platforms 44
3.7 Productivity of Concurrent, Parallel and Stream Programming Platforms 46
3.8 Summary of Concurrent and Parallel Programming Platforms 48
3.9 Productivity of Compilation and Runtime Platforms . . 52
3.10 Efficiency of Compilation and Runtime Platforms 53
3.11 Adoption of Compilation and Runtime Platforms 54
3.12 Summary of Compilation and Runtime Platforms 55
3.13 Summary of the state of the art 57

6.1 Summary of the proposed solution 94
6.2 Summary of the perspective 97

Illustrations credits

XKCD 934 https://xkcd.com/934/ 11
Dennis Salvatier http://bit.ly/dennis-salvatier-superman . 14
Martin David http://bit.ly/martin-david-server 15
Robert Wucher http://bit.ly/robert-wucher-chip 18
Benoit Hediard http://bit.ly/benoit-hediard-software . . 24
Justin Mezzell http://bit.ly/justin-mezzell-curiosity . . 41
Grumpy Cat coffee company http://drinkgrumpycat.com . . . 68

CHAPTER 1

INTRODUCTION

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 a *Fluxional compiler to bring seamless shift from development productivity to 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. It allows the development team to focus on application specifications rather than technical optimizations and eases 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 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 initial monolithic approach to adopt a more efficient processing model

[1] `http://bit.ly/startup-failures`

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 explores 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 pipeline architecture.

This thesis contains two main contributions. The first contribution is an equivalence allowing to split a program into a pipeline of stages depending on a common memory store. The second contribution is an improvement from the first equivalence to enforce isolation between the stages of this pipeline. With these two contributions, this thesis present the implementation of a compiler to transform the modular representation of an application 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 the notions outlined in chapter 2 to help the reader understand better the context. It finally 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. Finally, chapter 6 concludes this thesis. It evaluates this implementation at the light of the previous works, and draws the possible perspectives.

CHAPTER 2

CONTEXT AND OBJECTIVES

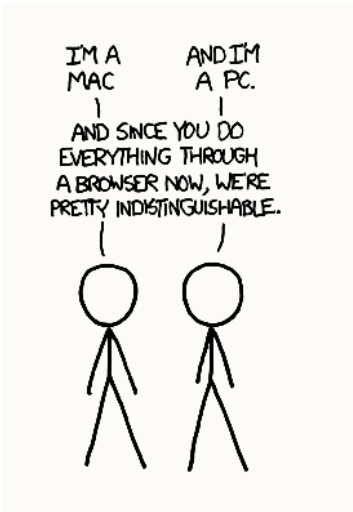
The web allows applications to grow a user base very quickly. The developement of a web application needs to follow this rapid pace to assure performance efficiency. But the languages often fail to grow with the project they initially supported very efficiently.

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, the Web browser started as a software product with extension capabilities that transformed it into a platform. 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 disruptive business model based on an instantaneous and free access for the user, while claiming value for their data. This thesis focuses not on this buisness model, but rather on the technologies that brought it.

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 adopt a language because it allows to quickly experiment and enter business 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 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://bit.ly/stroustrup-quote`

But this situation evolved drastically since. All web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime [56]. 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.

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 stimulated 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 [64]. It allows Javascript to develop richer applications inside the browser, from user interactions to network communications. The community released some frameworks to assist the development of these larger applications. Prototype[5] and DOJO[6] are early famous examples, and 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

[2] `http://bit.ly/coding-horror-view-source`

[3] True hipsters used Server-Side Javascript before it was cool. `http://bit.ly/mdn-server-side-javascript`

[4] `http://bit.ly/crockford-quote`

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

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

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

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

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

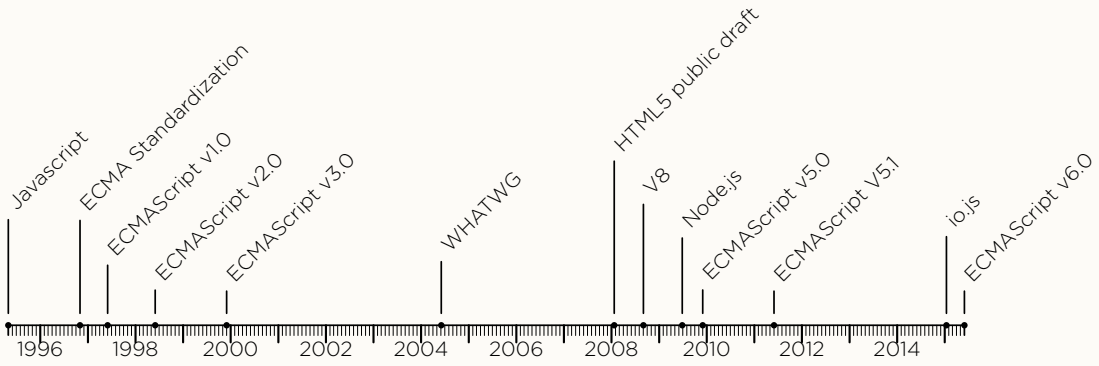


Figure 2.1 – Javascript timeline

is misleading, it is really about giving Javascript superpowers like geolocation, storage, audio, video, and many mores. The simultaneous releases 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 main events in the history of Javascript presented in the previous chapters are summariezed in figure 2.1.

×

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.

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

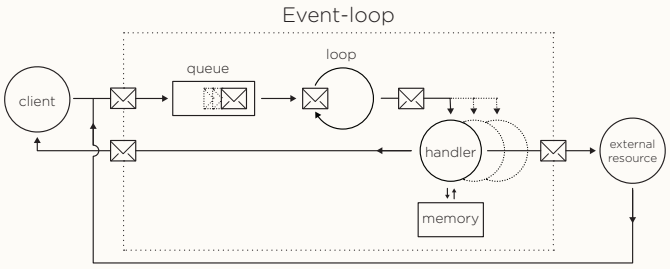


Figure 2.2 – Event-driven execution model

2.1.2 HIGHLY CONCURRENT WEB SERVERS

Since the web allows 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] [92]. 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.

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.

The event-driven execution model is presented in figure 2.2. At reception, each request from a client queues an event waiting to be

[11] http://bit.ly/cisco-connection-counter
[12] http://bit.ly/google-search-statistics
[13] http://bit.ly/c10k-problem
[14] http://bit.ly/c10m-problem

processed. A loop unqueues these events one at a time, and runs the appropriate handler to process them. To process an event, a handler can query external resources, which respond asynchronously by queuing additional events. For each query, the querying handler specifies a new handler - called a continuation - to process the additional event. Alternatively, a handler can respond directly to the client, ending this chain of asynchronous events.

This execution model allows high concurrency. It is required to respond to a high number of users simultaneously. Additionally, this concurrency needs to be scalable to adapt to the growth of audience.

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 between the tasks. The pipeline execution model avoid interdependencies between the tasks to assure their parallel execution.

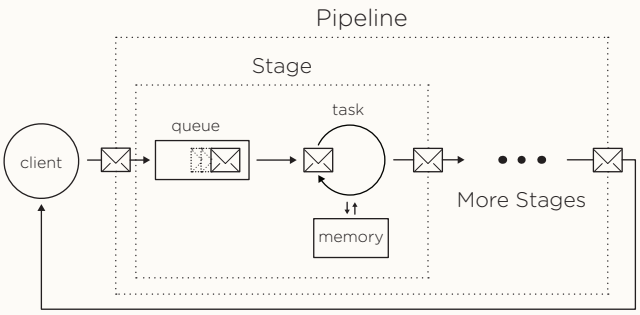


Figure 2.3 – Pipeline execution model

PIPELINE EXECUTION MODEL

The pipeline execution model, presented in figure 2.3, 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.

The execution model of each stage is organized in a similar fashion than the event-driven execution model 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 2.4. The isolation of memory in the pipeline execution model impacts the productivity of its programming model.

The event-driven requires a global memory to assure development productivity. Whereas the pipeline execution model assures the isolation between the tasks to assure efficiency. The former presents limited scalability, whereas the latter does not. This thesis argues that there exists an equivalence between the event-driven model and the pipeline execution model. Before introducing this equivalence, the next section details further the incompatibility between the two programming model and the resulting economical consequences.

2.2 AN ECONOMICAL PROBLEM

With Software as a Service (SaaS), the software industry is in charge of both development and execution of the software. Conducting the two at world wide scale is challenging, because they imply opposed economical constraints.

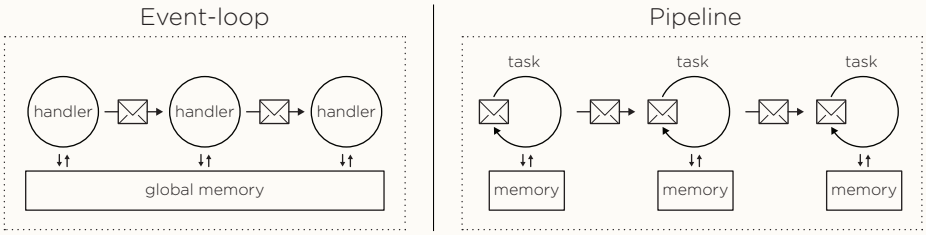


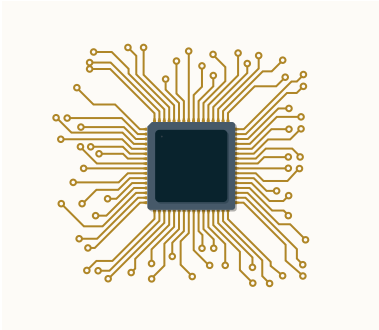
Figure 2.4 – Comparison of the two memory models

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 needs to revise its approach to meet these different constraints. Which leads to disruptions in the evolution of the application.

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.



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 the latter improves 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.

[15] `http://bit.ly/dennard-scaling`

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 is less productive. 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 solution proposed in this thesis would allow developers to iterate continuously on the implementation. They would focus progressively on productivity then efficiency.

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. The company identified that one of its 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 is to avoid the disruption in development.

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, which would 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. This thesis develops an equivalence to map these two models, despite their differences.

Both programming 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.

These two organizations are incompatible which results ruptures in the development. It represents additional development efforts and important costs. This thesis argues that it is possible to lift these efforts and costs. To do so, it proposes an equivalence between the two organizations to allow the development to be continuous. It briefly introduces this equivalence in the next paragraph, and details it further in the chapter 4 and 5.

In the beginning of a project, the team focuses on productivity, maintainability and evolution, discarding the efficient and scalable performance concerns. The team adopts the event-driven execution model and always sticks with the productive model. And as the project gather audience and the performance concerns become more and more critical. The equivalence allows to transform an application expressed in the event-driven execution model into the pipeline execution model. The generated pipeline expression allows the execution engine to adapt itself to any parallelism, from a single core, to a distributed cluster. Without giving the productive model up, the development team takes advantage of the different concerns of the two execution models, productivity and efficiency.

×

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

> *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 [160]

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.

The productivity of a platform is the degree to which developers can quickly produce new and modify existing softwares. It impacts the maintainability of the applications and relies on the modularity enforced by its platform. *75% of your budget is dedicated to software maintenance.*[1] 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

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

focusing on productivity are addressed in section 3.2, those 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, Productivity, Efficiency and Adoption. This section defines these tree criteria and their 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. To support productivity, a platform needs to enforce modularity directly in the design of applications. Productivity leads to maintainability.

Measuring productivity precisely is highly difficult. The measurement varies more between developers than between platforms [149]. Instead of measuring directly the productivity of a platform, this thesis infer productivity by measuring modularity.

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 [157]. And it reduces development time by allowing developers to implement different modules simultaneously [181, 26].

The criteria to define modules to improve productivity are high cohesion enforced by encapsulation and low coupling enforced by composition [157]. Cohesion defines how strongly the features inside a module are related. Coupling defines the strength of the interdependences between modules.

$\times$

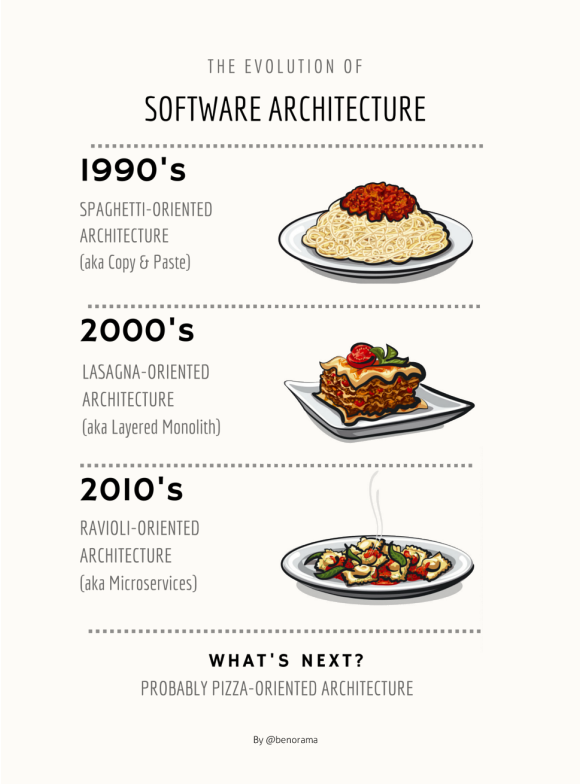
The criteria for productivity are the encapsulation and composition allowed by a platform. Encapsulation relies on the definition of boundaries, and the protection of data. Composition relies on higher-order programming and lazy evaluation. The next paragraphs define these requirements.

- Encapsulation (Boundary definition, Data protection)
  → increases Cohesion
- Composition (Higher-order programming, Lambda Expressions)
  → decreases Coupling

ENCAPSULATION

Modular Programming draws clear interfaces around a piece of implementation so that the execution remains enclosed inside [47]. At a fine level, it helps avoid spaghetti code [50], and at a coarser level, it structures the implementation [51] into modules, or layers.

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 [138, 165, 100]. 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.



COMPOSITION

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 [104], the Hollywood Principle [163], or Monads [176]. Higher-order programming help loosen coupling, thus improve productivity [83].

In languages allowing mutable state, lambda expressions are implemented with closure to preserve the lexical scope [162]. 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.

[2] http://bit.ly/oop-patterns

3.1.2 EFFICIENCY

The efficiency of a software project is the relation between the usage made of available resources and the delivered performances. 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 [36]. 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.

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 [7, 74]. 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 as immutable messages.

Because of the latency associated with message-passing, the atomicity of operations is challenged.

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.

GRANULARITY

If the operations access the state too frequently, the communication overhead of message passing exceeds the performance gains of parallelism. Whereas 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 more closely at a fine-grain level and more loosely 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 [76, 75, 130, 73]. 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 platforms 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. 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.

1 references section 3.2.1, the productivity focused platforms.
2 references section 3.2.2, their steering back toward efficiency.
3 references section 3.3.1, the efficiency focused platforms.
4 references section 3.3.2, their steering back toward productivity.
5 references section 3.4, the platforms with a balance between productivity and efficiency. Moreover, the platforms are rated for each criterion on a scale from 0 to 5: 0, 1, 2, 3, 4, 5.

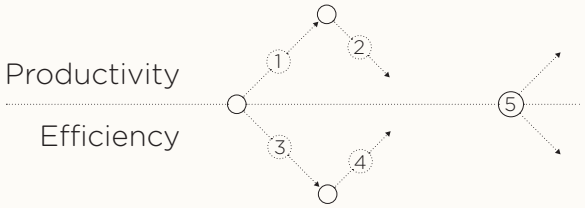


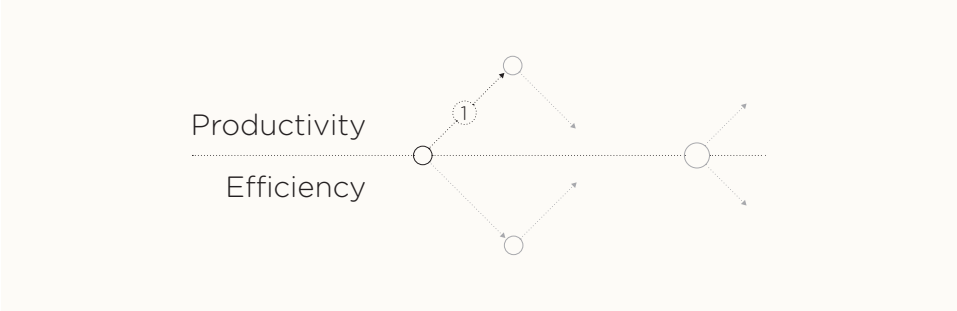
Figure 3.1 – Balance between Efficiency and Productivity

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 [157].

In order to improve and maintain a software system, it is important to holds in mind a mental representation of its implementation [153]. 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.

The modular programming paradigms is precisely designed for this purpose. It is presented in section 3.2.1 with the programming models oriented toward productivity. The implementations of these models are addressed in section 3.2.2. The consequences of modularity on performance are addressed in section 3.2.3. Finally, section 3.2.4 summarizes this three previous sections.



3.2.1 MODULAR PROGRAMMING

IMPERATIVE PROGRAMMING

Imperative programming is the very first programming paradigm, as it evolves directly from the hardware architectures. It allows to express the suite of operation to carry sequentially on the computing processor. Most imperative languages provide encapsulation with modules but not higher-order programming. The main implementations of Imperative Programming are Fortran, Algol, Cobol and C.

OBJECT ORIENTED PROGRAMMING

The very first Object-Oriented Programming (OOP) language was Smalltalk [66]. It defined the core concepts as message passing and encapsulation [3]. Nowadays, the emblematic figures in the software industry are C++ [159] and Java [68]. They provide encapsulation with Classes, and allow mutable structures for performance reasons. They recently introduced higher-order programming with lambda expressions.

FUNCTIONAL PROGRAMMING

The definition of pure Functional Programming resides in manipulating only expressions and replacing state mutability, with immutable message-passing. The absence of state mutability makes a function side-effect free, hence their execution can be scheduled in parallel. The most important pure Functional Programming languages are Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]. They provide encapsulation, higher-order programming and lazy evaluation.

[3] http://bit.ly/mail-alan-key-meaning-oop

MULTI-PARADIGM

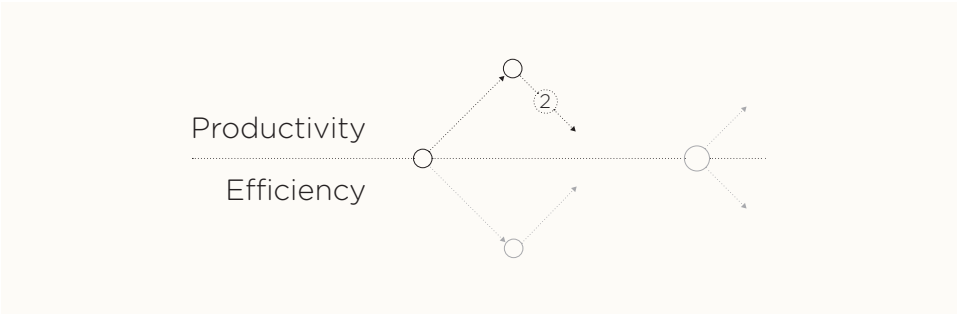
The functional programming concepts are also implemented in most mainstream languages along with mutable states and object-oriented concepts. Major recent programming languages now commonly present higher-order functions, including Java 8 and C++ 11. The main multi-paradigm languages are Javacript, Python, Ruby and Scala [135]. These multi-paradigms languages combine the different paradigms to help developer building applications that are more maintainable, and favorable to evolution [99, 172].

×

Thes different programming models present different approach to provide productivity, as recapped in table 3.1. The imperative programming model is less productive than the other, as it is the oldest one. The two following - Object-Oriented, and Functional Programming - adopted different approaches to improve upon it. And finally, the last one selected improvements from the two diverging approaches as the development practices evolved.

Model	Implementations	Composition	Encapsulation	→	Productivity
Imperative Programming	Fortran, Algol, Cobol and C	3	4		3
Object-Oriented Programming	C++ [159] and Java [68]	4	5		4
Functional Programming	Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]	5	4		4
Multi Paradigm	Javacript, Python, Ruby and Scala [135]	5	5		5

Table 3.1 – Productivity of Modular Programming Platforms



3.2.2 STEERING BACK TOWARD EFFICIENCY

As stated previously, adoption relies on productivity as well as efficiency. Industrial actors often supply resources to improve the efficiency of major productive languages. The performance improvement on the V8 Javascript execution engine are a good example[4]. The increasing adoption of Javascript can be explained both by its productivity, and by its increased efficiency.

COMMUNITY

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.

Online collaboration tools give an indicator of the number of developers and projects using certain languages. Javascript is the most used language on *Github*[5] and the most cited language on *StackOverflow*[6]. It represents more than 320,000 repositories on *Github*. The second language is Java with more than 220,000 repositories. It is cited in more than 960,000 questions on *StackOverflow* while the second is Java with around 940,000 questions. And according to a survey by *StackOverflow*, it is currently the language the most popular[7]. Moreover, the Javascript package manager, *npm*, has the most important and impressive package repository growth.

[4] `http://bit.ly/V8-optimization`
[5] the most important collaborative development platform.
[6] the most important Q&A platform for developers.
[7] `http://bit.ly/stackoverflow-developer-survey-2015`

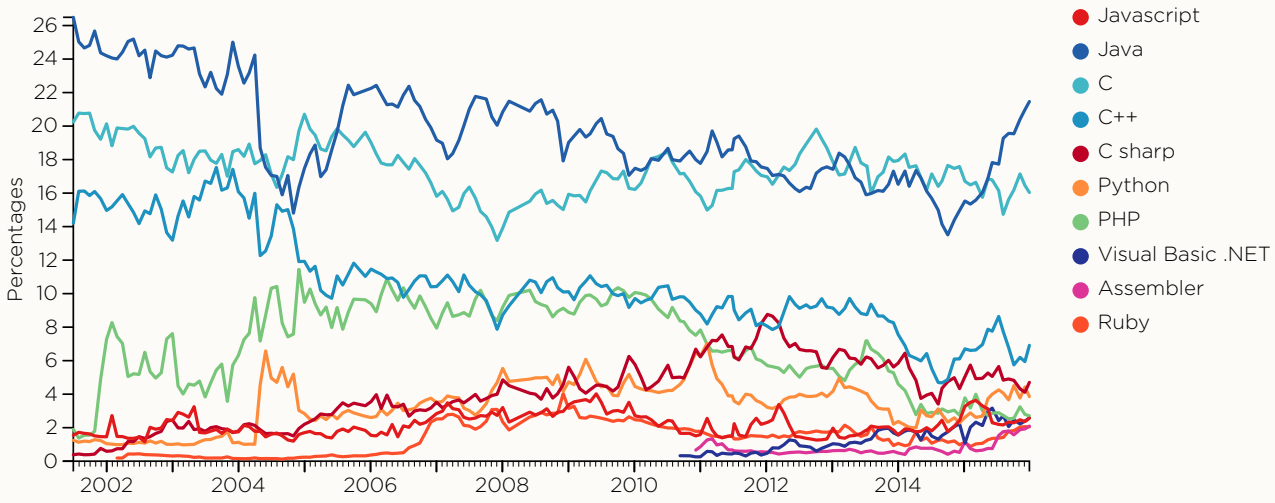


Figure 3.2 – TIOBE ranking

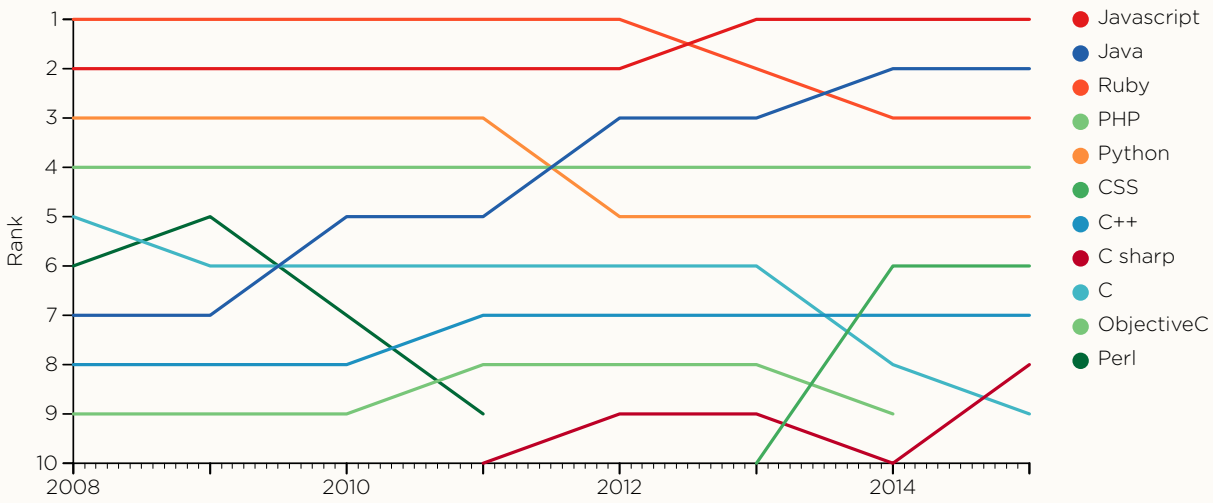


Figure 3.3 – Languages Ranks from number of Github projects

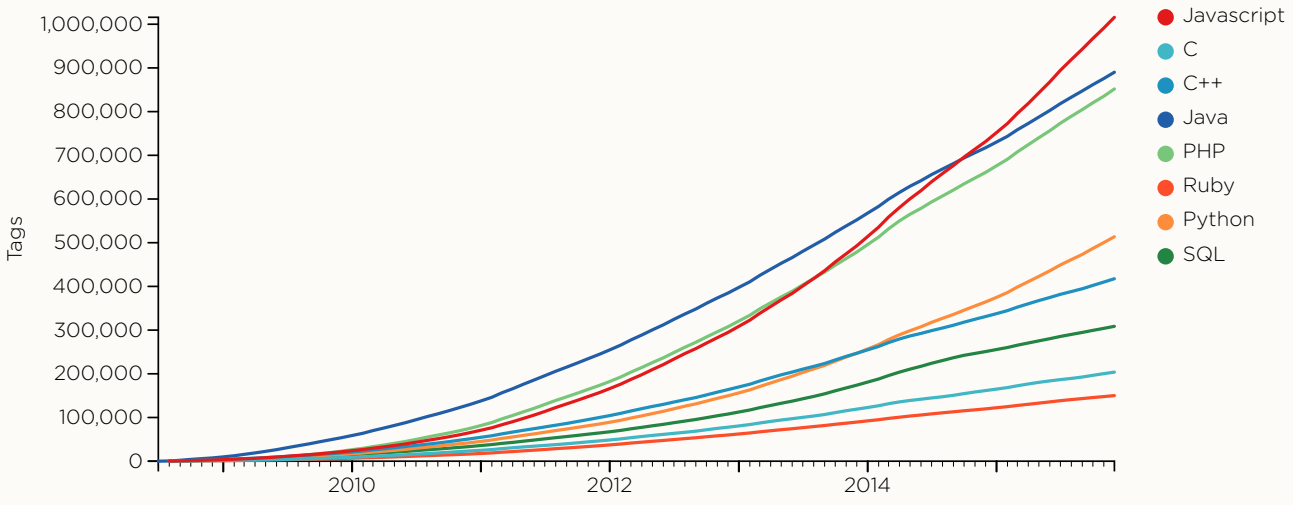


Figure 3.4 – StackOverflow Tags evolution

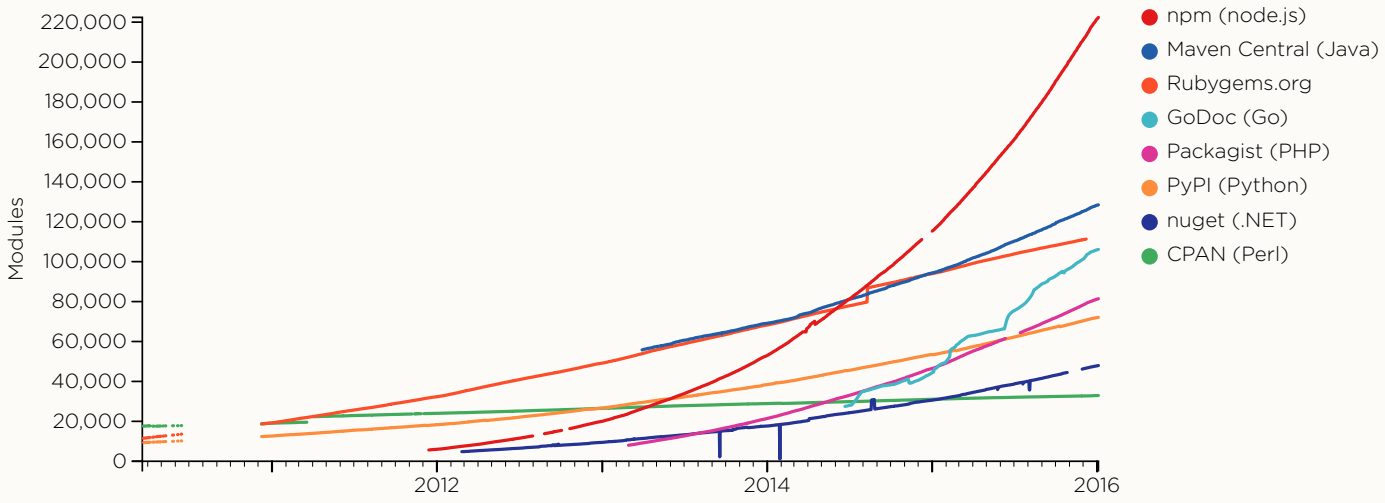


Figure 3.5 – Module Counts per package manager

INDUSTRY

The industrial actors avoid to disclose their activities believing it grants them an edge on the competition. The previous metrics represent the visible activity but are barely representative of the software industry. The trends on job opportunities give some additional hints on the situation. Javascript is the third most wanted skill, according to *Indeed*[8], right after SQL and Java.[9] Moreover, according to *breaz.io*[10], Javascript developers get more opportunities than any other developers. Javascript is increasingly adopted in the software industry.

×

Multi paradigm languages like Javascript are the most widely used by the community and the industry, as presented in table 3.2. OOP and Imperative Programming remains strong in the industry, but are slightly less active in the community. As detailed previously, the main OOP languages are steering toward the multi-paradigm approach. With lambda, for example. Functional Programming is not well represented. It comes mainly from the academy, and failed to penetrate the academy nor the industry.

Model	Implementations	Community support	Industrial need	→	Adoption
Imperative Programming	Fortran, Algol, Cobol and C	2	3		2
Object-Oriented Programming	C++ [159] and Java [68]	4	4		4
Functional Programming	Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]	1	2		1
Multi Paradigm	Javacript, Python, Ruby and Scala [135]	5	3		3

Table 3.2 – Adoption of Modular Programming Platforms

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

[9] `http://bit.ly/indeed-developer-jobs-comparator`

[10] *breaz.io* `https://breaz.io/` was recently acquired by *hired* `https://hired.com/`.

3.2.3 EFFICIENCY LIMITATIONS

Eventually, the presented languages are hitting a wall on their way to performance. They provide global memory abstraction on which to rely to assure encapsulation and composition. Functional programming relies on immutable message-passing. It might impact performance at a fine-grain level because of heavy memory usage. And the synchronization required by mutable state is often hard to develop with [2], or avoid parallelism [137, 110]. These results are recapped in table 3.3.

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Imperative Programming	Fortran, Algol, Cobol and C	4	1		1
Object-Oriented Programming	C++ [159] and Java [68]	4	1		1
Functional Programming	Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]	1	4		1
Multi Paradigm	Javacript, Python, Ruby and Scala [135]	4	1		1

Table 3.3 – Efficiency of Modular Programming Platforms

The only solution to provide performance efficiency is to combine mutable state at a fine-grain level and immutable state at a coarse-grain level [11]. That is to provide both synchronization and message-passing to allow parallelism. The platforms extending these languages with concurrent or parallel paradigms to improve efficiency are addressed in the next section.

3.2.4 SUMMARY

The platforms presented in this section recount the evolution of mainstream development toward productivity. From Imperative Programming to Multi-Paradigms, languages improved the development productivity. This improvement in productivity is steered by the community as well as the industry. The community is attracted by the most

[11] `http://bit.ly/joe-duffy-immutability-concurrency`

productive languages to quickly learn and start new projects. While the industry makes them more attractive by providing resources to improve their efficiency. The interaction of the industry and the community results in these mainstream languages heavily adopted. However, there is inherently a limitation to the efficiency of these productive languages.

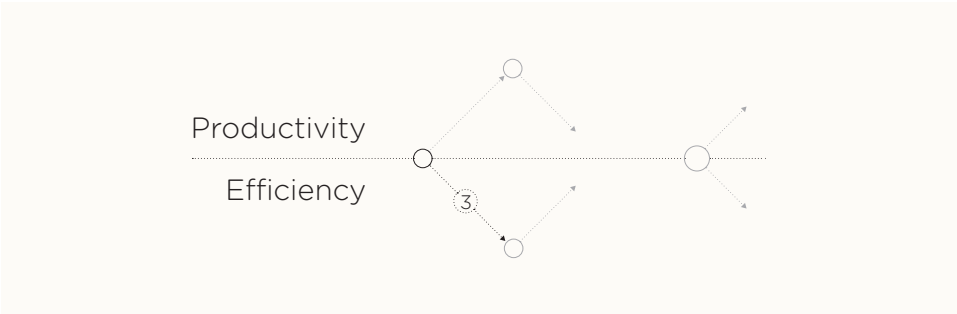
These languages are productive, but this efficiency limitation avoid an unanimous adoption, as summarized in table 3.4.

Model	Implementations	Productivity	Efficiency	Adoption
Imperative Programming	Fortran, Algol, Cobol and C	3	1	2
Object-Oriented Programming	C++ [159] and Java [68]	4	1	4
Functional Programming	Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]	4	1	1
Multi Paradigm	Javacript, Python, Ruby and Scala [135]	5	1	3

Table 3.4 – Summary of Modular Programming Platforms

3.3 EFFICIENCY FOCUSED PLATFORMS

To cope with the limitations the previous section concludes on, both the academia and the industry proposed solutions discarding productivity to focus on efficiency. Among these solutions, the concurrent and parallel programming paradigms are presented in section 3.3.1 They are evaluated against the same criteria than the previous section - Productivity, Efficiency and Adoption. This evaluation illustrates the impact of this shift of focus on the adoption of these platforms, as presented in section 3.3.2. Finally, section 3.3.3 presents the limitations of parallelism on productivity.



3.3.1 CONCURRENCY

Web servers need to be able to process lots 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.

CONCURRENT PROGRAMMING

> *“No matter how great the talent or efforts, some things just take time. You can’t produce a baby in one month by getting nine women pregnant.”*
>
> — Warren Buffett[12]

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.

[12] `http://bit.ly/warren-buffet-quote`

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 event-driven programing model relies on cooperative scheduling, and the multi-threading programming model relies on preemptive scheduling.

EVENT-DRIVEN PROGRAMMING

In the event-driven execution model, developers explicitely split the application into several concurrent tasks called handlers. The execution model schedules these handlers sequentially with the queue of events, to assure exclusivity on shared resources, and cooperatively to assure atomicity. Moreover, they communicates with the resources asynchronously to avoid waiting. A handler yields execution to the next handler to complete the communication.

Promises were introduced as an help to nicely chain these concurrent handlers [117]. A promise is a placeholder for a future result allowing to defer operations for when the result is available. It layouts the concurrent handlers in chains, similarly to a pipeline.

This execution model is very efficient for highly concurrent applications as it avoids contention. Several platform rely on this execution model, like TAME [110], Node.js[13] and Vert.X[14]. As well as some web servers like Flash [137], Ninja [69] thttpd[15] and Nginx[16].

Though, the event-driven model is limited in performance because of the global memory. The handlers cannot be scheduled in parallel. Lock-free data-strincture intends 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 [112, 88, 86, 87, 9]. They provide concurrent implementations of basic data-structures such as linked list [174, 168], queue [161, 180], tree [143] and stack [85]. They rely on instructions provided by transactional memories [81] that combine read and write instructions.

Lock-free Data-structures allow parallelization of the concurrent tasks, but are mostly unavailable on common hardware. Multi-threading intends to provide parallelization with software protection.

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

MULTI-THREADING PROGRAMMING

Threads are small execution containers sharing the same memory execution context within an isolated process [51], and scheduled in parallel with fork/join instructions [144, 61, 114]. They execute statements synchronously and are preempted to avoid blocking the progression of other threads. Without protection, the preemption breaks the atomicity and exclusivity of memory accesses. To assure atomicity and exclusivity, hence invariance, multi-threading programming models provide protection mechanisms, such as semaphores [48], guarded commands [49], guarded region [80] and monitors [94].

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 [2].

HYBRID MODELS

Hybrid models intends to join the productivity advantage of synchronous execution from thread, with the efficiency advantage of asynchronous execution from events-driven models. The implementations of hybrid models are libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]. For example, cooperative threads, or fibers, avoid splitting the execution into atomic tasks nor use protection 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, it wasn't adopted in practice. Hiding the yielding points effectively hide the preemption. Developers cannot protect atomicity of operations, which increases the likeliness of race conditions [17].

LIMITATION OF CONCURRENT PROGRAMMING

The presented concurrent programming paradigms assure sequentiality of execution within a task and the causal ordering between tasks. Multi-threading imposes heavy protection mechanisms and fails to avoid contention because of the preemptive scheduling. Hybrid models tries to improve over multi-threading using cooperative scheduling. But the synchronous execution imposed by these two models 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 [111, 146]. And it is easier for developer to define causal scheduling than to assure the consistency of memory with protection mechanisms. But because of the lack of memory isolation, the concurrent tasks are not scheduled in parallel. It impacts efficiency, as detailed in table 3.5. Parallel

[17] `http://bit.ly/deciphering-glyph-unyielding`

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

PARALLEL PROGRAMMING

The Flynn's taxonomy [57] 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) [12, 44, 45] and Multiple Program Multiple Data (MPMD) [31, 29]. SPMD defines a single program replicated on many processing units [41, 103, 30] – it is roughly derived from the multi-threading programming model presented above. While MPMD defines multiple parallel tasks in the implementation [70, 59, 58]. The two major theoretical models for MPMD parallel programming are the Actor Model and Communicating Sequential Processes.

ACTOR MODEL

The Actor Model allows to express the causal ordering of computation as a set of parallel actors communicating by asynchronous messages [89, 90, 37]. In reaction to a received message, an actor can create other actors, send messages, and choose how to respond to the next message. In reality, the communication are too slow compared to execution to be synchronous, and are subject to various faults and attacks [113]. The Actor model takes these physical limitations in account [91].

COMMUNICATING SEQUENTIAL PROCESSES

Similar works include the Communicating Sequential Processes (CSP) [93, 23], and the Kahn Networks [106, 107]. Coroutines are autonomous programs which communicate with adjacent modules as if they were input and output subroutines [40]. It defines a pipeline to implement multi-pass algorithms.

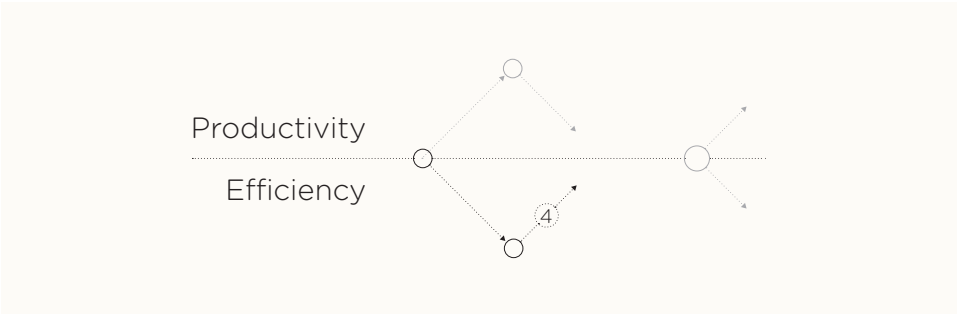
The difference between Actors and Coroutines lie in the definition of the communication. Actors send messages to named actors, whereas Coroutines communicate through named channels.

SUMMARY OF CONCURRENT AND PARALLEL PROGRAMMING MODELS

Compared to parallel programming, the concurrent programming models are unable to correctly isolate the concurrent tasks and communicate by message-passing to allow parallelism. Hence the parallel programming models are more efficient than the concurrent programming models, as detailed in table 3.5.

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Event-driven programming	TAME [110], Node.js[18] and Vert.X[19]	5	2		2
Lock-free Data-Structures	linked list [174, 168], queue [161, 180], tree [143] and stack [85]	5	2		2
Multi-threading programming	semaphores [48], guarded commands [49], guarded region [80] and monitors [94]	5	1		1
Hybrid Models	libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]	5	1		1
Actor Model	Erlang [11, 129, 10], Scala Actors [78], Akka[20] and Play[21]	5	5		5
Communicating Sequential Processes	Go[22]	5	5		5

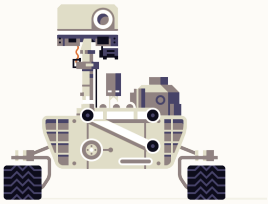
Table 3.5 – Efficiency of Concurrent and Parallel Programming Platforms



3.3.2 STEERING BACK TOWARD PRODUCITIVITY

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 Rovers or some banking systems are about 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 trades productivity for efficiency, the less support it receives from the community.



CONCURRENT PROGRAMMING

Most programming language implementations supports concurrent programming. Either with multi-threading or event-driven programming. These two are highly adopted by both the industry and the community, as presented in table 3.6.

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.

PARALLEL PROGRAMMING

There exists several platforms directly inspired by the actors model, like Erlang [11, 129, 10], Scala Actors [78], Akka[23] and Play[24]. 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 [11, 129, 10]

[23] http://akka.io/
[24] https://www.playframework.com/

There are as well other platforms inspired by other theoretical model, like Go[25], 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 strongly passionated but small communities.

Indeed, it is difficult for developers to manage the superposition of these two organizations, tasks and modules. The organization in independent tasks is hardly compatible with the modular organization presented in the previous section. This superposition makes these platforms accessible only to an elite in the industry supporting it. To mitigate this difficulty, various platforms help organizing the tasks, or adopt the same granularity for modules and tasks to fit the two organizations.

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 [38, 46, 124, 67]. Developers focus on their problem and delegate the communications to specialized skeletons. 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 [46].

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[26], EJB[27] and Spring[28]. It allows to adjust the granularity of tasks to help developers to better fit the tasks organization with the modular organization [1].

More recently, Microservices are tackling the same challenge on the web, with a finer granularity [55, 60, 128]. AN example of Microservice platform is Seneca[29]. Microservices are very recent, and it is difficult to asses their usage in the community nor the industry. But they seem to be increasingly adopted, both in the industry and in the community.

$\times$

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. This requirement imposes specific challenges for the platforms.

[25] https://golang.org/
[26] https://www.osgi.org/developer/specifications/
[27] http://www.oracle.com/technetwork/java/javaee/ejb/index.html
[28] http://projects.spring.io/spring-framework/
[29] http://senecajs.org/

STREAM PROCESSING SYSTEMS

DATA-STREAM MANAGEMENT SYSTEMS

Database Management Systems (DBMS) historically processed large volumes of data, and they naturally evolved into Data-stream Management System (DSMS) to process data streams as well. Because of this legacy, 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 [101, 185], Apache Hive [167], Timestream [140], Shark [182].

PIPELINE STREAM PROCESSING

On the other hand, the pipeline architecture is directly inspired by the Actors Model and CSP. It organizes an application as a network of event-driven stages connected by explicit queues, the output of one feeding the input of the next [179]. The event-driven paradigm of a stage is similar to work like Ninja [69] and Flash [137] 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 SEDA [179], StreaMIT [166], Spidle [39], Aspen [173], Pig Latin [136], MEDA [79], CBP [118] and S4 [131] designed at Yahoo, Piccolo [139], Comet [84], Nectar [72], SEEP [125], Legion [15], Spark Streaming [187], Naiad [127] designed at Microsoft, Millwheel [4] designed at Google, Halide [142], Storm [169] by Twitter, SDG [54], Regent [154] and Neptune [25].

×

Parallel programming emerges mainly from industrial needs and academic research but is barely supported by the community. The implementations improve efficiency. But their weak productivity prevents their adoption by the community. As summarized in table 3.6, the event-driven programming model is the best candidate for a concurrent programming model regarding adoption. It is supported by the community, and respond to concrete needs in the industry.

Model	Implementations	Community support	Industrial need	→	Adoption
Event-driven programming	TAME [110], Node.js[30] and Vert.X[31]	5	5		5
Lock-free Data-Structures	linked list [174, 168], queue [161, 180], tree [143] and stack [85]	0	2		0
Multi-threading programming	semaphores [48], guarded commands [49], guarded region [80] and monitors [94]	3	5		3
Hybrid Models	libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]	0	0		0
Actor Model	Erlang [11, 129, 10], Scala Actors [78], Akka[32] and Play[33]	1	5		1
Communicating Sequential Processes	Go[34]	1	5		1
Skeleton	MapReduce	2	5		2
Service Oriented Architecture	OSGi[35], EJB[36] and Spring[37]	3	4		3
Microservices	Seneca[38]	3	3		3

Table 3.6 – Adoption of Concurrent and Parallel Programming Platforms

3.3.3 PRODUCTIVITY LIMITATIONS

Parallel programming requires the organization of execution and memory into independent tasks. This independence imposes different granularity of state accessibility. At a fine level, the state is shared, while at a coarser level, it is isolated. It avoids higher-order programming. Hence, it limits the composition of modules and negatively impacts productivity.

Without this composition between modules, parallel programming forces to develop two mental representations – one for the module organization and one for the task organization – or to abandon the module organization and productivity altogether. It makes parallel programming accessible only to an elite of developers that are able to be productive despite the two mental representations, as presented in table 3.7.

Model	Implementations	Composition	Encapsulation	→	Productivity
Event-driven programming	TAME [110], Node.js[39] and Vert.X[40]	5	5		5
Lock-free Data-Structures	linked list [174, 168], queue [161, 180], tree [143] and stack [85]	5	5		5
Multi-threading programming	semaphores [48], guarded commands [49], guarded region [80] and monitors [94]	4	4		4
Hybrid Models	libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]	4	4		4
Actor Model	Erlang [11, 129, 10], Scala Actors [78], Akka[41] and Play[42]	2	2		2
Communicating Sequential Processes	Go[43]	2	2		2
Data Stream System Management	DryadLINQ [101, 185], Apache Hive [167], Timestream [140], Shark [182]	2	3		2
Pipeline Stream Processing	SEDA [179], StreaMIT [166], Spidle [39], Aspen [173], Pig Latin [136], MEDA [79], CBP [118] and S4 [131] designed at Yahoo, Piccolo [139], Comet [84], Nectar [72], SEEP [125], Legion [15], Spark Streaming [187], Naiad [127] designed at Microsoft, Millwheel [4] designed at Google, Halide [142], Storm [169] by Twitter, SDG [54], Regent [154] and Neptune [25]	2	3		2

Table 3.7 – Productivity of Concurrent, Parallel and Stream Programming Platforms

3.3.4 SUMMARY

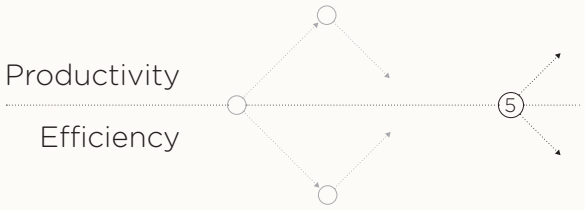
The parallel programming platforms presented in this section focus more on efficiency than productivity. Whereas the concurrent programming platforms feature better productivity. But no platform features both productivity and efficiency, as presented in table 3.8. Moreover, this lack of versatility impacts adoption. These platforms remains inaccessible for the community. Exception for Event-driven and Multi-threading programming which are among the productive ways to achieve concurrency.

Model	Implementations	Productivity	Efficiency	Adoption
Event-driven programming	TAME [110], Node.js[44] and Vert.X[45]	5	2	5
Lock-free Data-Structures	linked list [174, 168], queue [161, 180], tree [143] and stack [85]	5	2	0
Multi-threading programming	semaphores [48], guarded commands [49], guarded region [80] and monitors [94]	4	1	3
Hybrid Models	libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]	4	1	0
Actor Model	Erlang [11, 129, 10], Scala Actors [78], Akka[46] and Play[47]	2	5	1
Communicating Sequential Processes	Go[48]	2	5	1
Data Stream System Management	DryadLINQ [101, 185], Apache Hive [167], Timestream [140], Shark [182]	2	5	1
Pipeline Stream Processing	SEDA [179], StreaMIT [166], Spidle [39], Aspen [173], Pig Latin [136], MEDA [79], CBP [118] and S4 [131] designed at Yahoo, Piccolo [139], Comet [84], Nectar [72], SEEP [125], Legion [15], Spark Streaming [187], Naiad [127] designed at Microsoft, Millwheel [4] designed at Google, Halide [142], Storm [169] by Twitter, SDG [54], Regent [154] and Neptune [25]	2	5	1

Table 3.8 – Summary of Concurrent and Parallel Programming Platforms

3.4 COMPROMISE BETWEEN PRODUCTIVITY AND EFFICIENCY

The platforms previously presented focus on productivity or efficiency. The previous section concludes that favoring 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 to trigger a virtuous circle of adoption. Some platforms feature an abstraction of the task organization to allow developers to focus on the modular organization to keep both productivity and efficiency. This abstraction happens either at compile time or at runtime.



3.4.1 ABSTRACTION OF TASKS ORGANIZATION

COMPILERS

> *"It is a mistake to attempt high concurrency without help from the compiler."*
>
> — R. Behren, J. Condit, E. Brewer [17]

The idea to bridge the gap between the tasks and modules organizations dates from the initial paper that presented this gap, in 1972 by D. Parnas [138]. However, the implementation of this idea remains a work in progress.

PARALLELISM EXTRACTION

To extract parallelism from a sequential implementation, a compiler needs to identify the commutative operations to parallelize their executions [148, 36]. The parallelization of loop iterations has been thoroughly studied [123, 6, 34, 13, 141], particularly with the polyhedral compilation method [14]. Examples of polyhedral compilers are AlphaZ [186], Polly [71] and GRAPHITE [170]. To improve performance gains outside of loops, some compilers identify parallelism in the data-flow representation on the whole program [16, 27, 115].

Data processing applications [54] such as web services [150] are often already organized as data-flow. In higher-order programming, continuous passing style and promises encourage this data-flow organization. However, the mutable closures required for higher-order programming remains a challenge to parallelize because they rely on a globally shared memory [82, 132, 122]. To extract parallelism, compilers rely on static analysis or notations from the developers.

STATIC ANALYSIS

Compilers analyze the source code of a program to detect commutative operations in the control flow [5]. The point-to analysis is a static analysis method. It identifies multiple symbolic names pointing to the same memory location. That is called aliasing. Hence it idenfities side-effects [8, 102, 156, 178] between operations, which allows to infer their commutativity. Another method, the abstract interpretation, is to interpret the possible path of executions with virtual inputs. It allows to statically reason on the behavior of dynamic programs [119, 155, 63, 77, 145, 62, 19]. It is successfully used for security applications to detect malicious scripts, or obfuscate code [97, 105, 184, 120, 35, 52]

However, these static analysis methods remains often too imprecise, and expensive for the performance gain to be profitable in dynamic languages, such as Javascript [152]. 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 [175, 54]. Such annotations are especially relevant for accelerators such as GPUs or FPGAs, because the development effort yields huge performance improvements [164]. Examples of such compilers are OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96].

COMPILATION LIMITATIONS

For dynamic languages like Javascript, the static analysis is not sufficient to correctly infer the independence of tasks to parallelize them. Parallel compilers often fall back on relying on annotations provided by developers. Hence, the burden of detailing the tasks and memory organizations falls back to the developer. It impacts productivity and adoption.

RUNTIMES

At runtime, the uncertainties on the independence of tasks are resolved. It allows analysis precise enough to detect and distribute the commutative operations.

PARTITIONED GLOBAL ADDRESS SPACE

The Partitioned Global Address Space (PGAS) provides a uniform access on a distributed memory 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 provides 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 implementations of the PGAS model are CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109].

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 [150, 151]. 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.

PRODUCTIVITY AND EFFICIENCY

The platforms presented in this section intend to merge both productivity and efficiency in a single platform by bringing parallelism to productivity languages. Because they are based on productivity languages, they feature decent encapsulation, but they limit the use of higher-order programming between tasks to allow their isolation. Hence, it degrades composition, as presented in table 3.9.

Despite worse productivity, this parallelization bring good efficiency, as presented in table 3.11.

Model	Implementations	Composition	Encapsulation	→	Productivity
Polyhedral Compilers	AlphaZ [186], Polly [71] and GRAPHITE [170]	2	4		2
Annotation Compilers	OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96]	3	4		3
Partitioned Global Address Space	CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109]	2	4		2
Dynamic Distribution	Leda [150, 151]	2	4		2

Table 3.9 – Productivity of Compilation and Runtime Platforms

Model	Implementations	Fine-grain level synchronization	Coarse-grain level message passing	→	Efficiency
Polyhedral Compilers	AlphaZ [186], Polly [71] and GRAPHITE [170]	5	4		4
Annotation Compilers	OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96]	5	4		4
Partitioned Global Address Space	CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109]	4	4		4
Dynamic Distribution	Leda [150, 151]	4	4		4

Table 3.10 – Efficiency of Compilation and Runtime Platforms

3.4.2 LIMITATION

The platforms presented in this section come from the need of the industry to reduce the development commitment required for efficiency. However, these platforms respond exclusively to academic or industrial needs, and are barely supported by the community, as presented in table 3.11.

The balance between efficiency and productivity is not sufficient for a community of passionate to arise. To be largely adopted, the platforms need grow with its community. That implies allowing novices to start learning and experimenting at small scale. It incites the community to start projects, and grow them organically to build businesses. The context of web development is particularly adapted for this growth.

Model	Implementations	Community support	Industrial need	→	Adoption
Polyhedral Compilers	AlphaZ [186], Polly [71] and GRAPHITE [170]	0	3		0
Annotation Compilers	OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96]	2	3		2
Partitioned Global Address Space	CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109]	1	3		1
Dynamic Distribution	Leda [150, 151]	0	1		0

Table 3.11 – Adoption of Compilation and Runtime Platforms

3.4.3 SUMMARY

This section observed that a platform cannot trade productivity against efficiency without massively losing the community required to trigger the adoption, as detailed in table 3.12. Indeed, in both the static compilation approach and the dynamic runtime approach, reducing productivity eventually avoids the community to learn, and tweak with

the platform. Hence, as the adoption cannot rise from the community, it breaks the reinforcing loop between industry and community.

Yet, the dynamic runtime approach has not been exhaustively explored.

Model	Implementations	Productivity	Efficiency	Adoption
Polyhedral Compilers	AlphaZ [186], Polly [71] and GRAPHITE [170]	2	4	0
Annotation Compilers	OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96]	3	4	2
Partitioned Global Address Space	CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109]	2	4	1
Dynamic Distribution	Leda [150, 151]	2	4	0

Table 3.12 – Summary of Compilation and Runtime Platforms

3.5 DISCONTINUOUS DEVELOPMENTS

The previous sections presented a broad view of platforms and their balance between productivity and efficiency. It established that the platforms favoring one eventually sacrifice the other. Moreover, the adoption of these platforms proves that none of these compromises are sustainable. Indeed, as presented in table 3.13, no platforms provides productivity, efficiency and adoption.

Model	Implementations	Productivity	Efficiency	Adoption
Imperative Programming	Fortran, Algol, Cobol and C	3	1	2

Object-Oriented Programming	C++ [159] and Java [68]	4	1	4
Functional Programming	Scheme [147], Miranda [171], Haskell [98] and Standard ML [126]	4	1	1
Multi Paradigm	Javacript, Python, Ruby and Scala [135]	5	1	3
Event-driven programming	TAME [110], Node.js[49] and Vert.X[50]	5	2	5
Lock-free Data-Structures	linked list [174, 168], queue [161, 180], tree [143] and stack [85]	5	2	0
Multi-threading programming	semaphores [48], guarded commands [49], guarded region [80] and monitors [94]	4	1	3
Hybrid Models	libasync [42], InContext [183], Fibers [2], Capriccio [18] and Monadic hybrid concurrency [116]	4	1	0
Actor Model	Erlang [11, 129, 10], Scala Actors [78], Akka[51] and Play[52]	2	5	1
Communicating Sequential Processes	Go[53]	2	5	1
Data Stream System Management	DryadLINQ [101, 185], Apache Hive [167], Timestream [140], Shark [182]	2	5	1

[49] https://nodejs.org/en/
[50] http://vertx.io/
[51] http://akka.io/
[52] https://www.playframework.com/
[53] https://golang.org/

Pipeline Stream Processing	SEDA [179], StreaMIT [166], Spidle [39], Aspen [173], Pig Latin [136], MEDA [79], CBP [118] and S4 [131] designed at Yahoo, Piccolo [139], Comet [84], Nectar [72], SEEP [125], Legion [15], Spark Streaming [187], Naiad [127] designed at Microsoft, Millwheel [4] designed at Google, Halide [142], Storm [169] by Twitter, SDG [54], Regent [154] and Neptune [25]	2	5	1
Polyhedral Compilers	AlphaZ [186], Polly [71] and GRAPHITE [170]	2	4	0
Annotation Compilers	OpenMP [43], OpenCL [158], CUDA [134], Cg [121], Brook [24] and Liquid Metal [96]	3	4	2
Partitioned Global Address Space	CoArray Fortran [133], X10 [33], Unified Parallel C [65], Chapel[28], OpenSHMEM [32], Kokko [53], UPC++ [188], RAJA [95], ACPdl [3], HPX [108, 109]	2	4	1
Dynamic Distribution	Leda [150, 151]	2	4	0

Table 3.13 – Summary of the state of the art

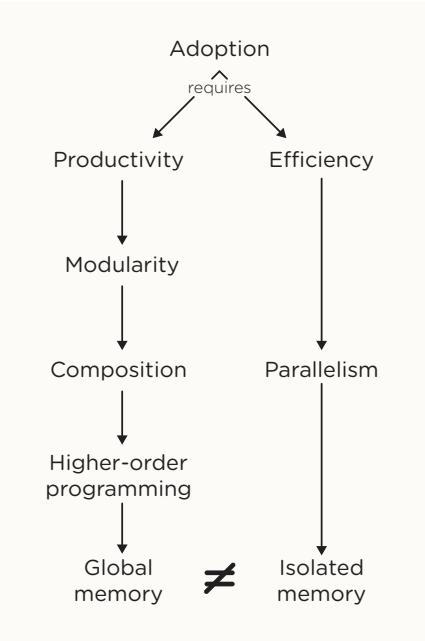
×

This chapter concludes on similar ground than the 1972 paper from D. Parnas [138]. Productivity requires modularity through encapsulation and composition. It requires higher-order programming which relies on a global memory abstraction as explained in section 3.2.3. Whereas efficiency requires a balance between fine-grain level shared state with synchronization and coarse-grain level independence with message-passing. This discontinuity between fine-grain level and coarse-grain level avoids the global memory abstraction, hence productivity. The absence of a global memory abstraction reserves efficient platforms for an elite of developers. No platform can support simultaneously productivity and efficiency. Nonetheless, a platforms needs to be adopted both by the industry and the community to be sustainable. D. Parnas then suggested the use of compilation techniques to bridge the gap between these two extremes.

However, more than this problem of immediate incompatibility, the problem holds on the evolution of the implementation. No platform is

able to follow a project from the early beginning until the industrial maturation of the project. All the platforms tends to be stuck in a compromise between these two goals, and cannot follow the evolution required for this compromise. These compromises are rigidly defined, while the need of the application is constantly evolving. They lack the possibility to follow the organic evolution of a project. Therefore, a project needs to change platform to change its priority, which leads to economical consequences.

To avoid these consequences, platforms would need to support productivity to allow the community to experiment, and organically start projects. And then continuously shift toward efficiency as the project evolves, and requires it. This thesis now explores this possibility.



CHAPTER 4

SEAMLESS SHIFT FROM PRODUCTIVITY TO EFFICIENCY

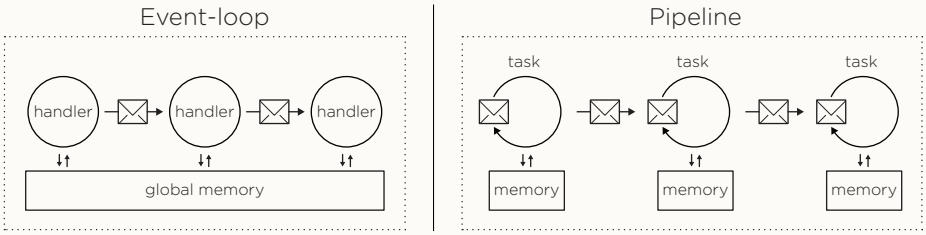


Figure 4.1 – Comparison of the two memory models

The evolution of the economical constraints of a web application requires to continuously shift from productivity to efficiency. The incompatibility between the two organizations forces platforms to propose a compromise between the two. And it makes it impossible for these platforms to follow the evolution of a web application. Hence, it 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.

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. Developers start a project without compromises on productivity. The language targets an event-driven execution model. Then, they continuously transform their implementation to target the more efficient pipeline architecture.

Node.js implements Javascript, a productivity language, with an event driven execution model to implement web applications with decent performances. However, the performance of this execution model is limited by the sequentiality of execution required to preserve exclusivity of memory accesses. The pipeline execution model overstep this performance limitation. It enforces memory isolation between stages allowing the parallel execution required for efficiency. But this isolation limits the productivity because it avoids higher-order programming. The difference between the two execution models is reminded in figure 4.1, from figure 2.4.

Despite this difference, these two execution models present an interesting similarity. They both organize the execution as a sequence of tasks causally scheduled. This thesis proposes an equivalence between these two execution models based on this similarity. Following this equivalence, it proposes a compiler that distributes the global memory into isolated stages of the pipeline. Concretely, it transforms a

mono-threaded, event-driven application to run on a parallel pipeline architecture.

4.1.1 CONTINUOUS DEVELOPMENT

This transformation allows a continuity of compromises between productivity and efficiency to seamlessly follow the shift of focus during development.

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 distributes the global memory into isolated stages as much as possible. It allows developers 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.

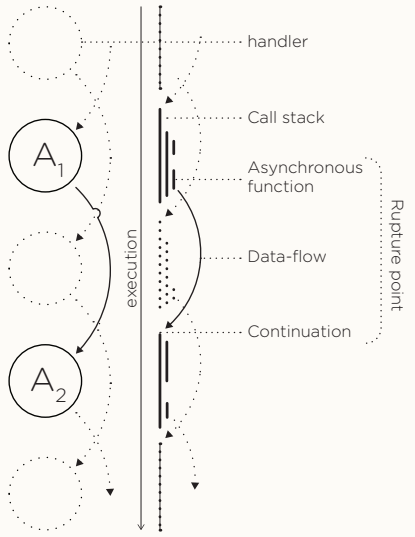


Figure 4.2 – Rupture point

4.1.2 EQUIVALENCE

The equivalence between these two execution models is broken down in two steps. The first step identifies the stages in the control flow, by detecting rupture points between them. The second step enforces isolation of memory between these stages, and replaces synchronization with message passing to preserve the invariance.

EXECUTION INDEPENDENCE

In the pipeline architecture, each stage has its own thread of execution independent from the others. Whereas in the event-driven execution model, the handlers 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 illustrated in figure 4.2. Indeed, a handler holds the execution until its call stack is empty, when all synchronous function calls terminates. Only then it yields the execution to the event-loop, which schedule the next handler. A rupture point separates the call stacks of two handlers.

For simplification, the figures 4.2 illustrates a rupture point with only two interleaving chains of handlers but there could be many more. Each chain is roughly comparable to a thread in multi-thread programming. The two chains in figure 4.2 are
→ (A1) → (A2) → and
◌ → ◌ → ◌.
And in figures 4.3, 4.4, 4.5 and 4.6, the two chains - A and B - are
→ (A1) → (A2) → and
◌ → (B1) → ◌.

RUPTURE POINTS

A rupture point is a call of a loosely coupled function. It is an asynchronous call without subsequent synchronization with the caller. This asynchronous function call is equivalent to a data-flow between two stages in the pipeline architecture. The parent sends a message to the child handler containing the result of the asynchronous function call it initiated. The event-driven execution model expects callback to send the message and continue the execution once the asynchronous call completes.

CALLBACKS, LISTENERS AND CONTINUATIONS

A callback is a function passed as a parameter to a function call. It is not inherently asynchronous. Only two kinds of callbacks are asynchronous, listeners and continuations. They are invoked to continue the execution with data not yet available synchronously, in the caller context. Listeners listen to a stream of events, hence are invoked multiple times. Whereas continuations are invoked only once to continue the execution after an asynchronous operation completes. 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. 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.

The rupture points are further detailed in section 5.2.1.

The detection of rupture points allows to retrieve the data-flow and the stages for a pipeline architecture from the implementation following the event-loop model. The implementation of this detection is fully addressed in the next chapter, in sections 5.1.3 and 5.2.1. However, these stages still require a global memory to communicate. They can't be executed in parallel without breaking the invariance.

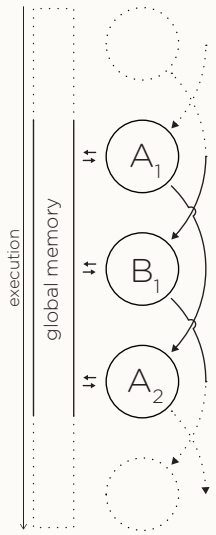


Figure 4.3 – Sequential scheduling

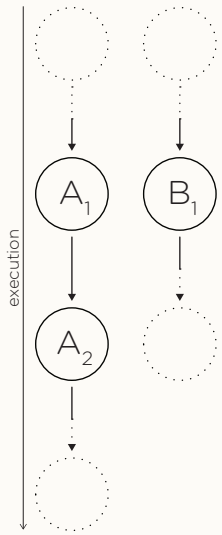


Figure 4.4 – Causal scheduling

INVARIANCE

A global memory requires the sequential scheduling of handlers to assure exclusivity of access, as illustrated in figure 4.3. The global memory is the only reason for this sequential execution. Yet, the causal scheduling of tasks is sufficient to assure the correctness of the execution.

Message passing only requires causal scheduling of handlers which allows the parallelism of the two chains, as illustrated in figure 4.4. If the handlers didn't rely on the global memory, they could be executed in parallel, as long as their causalities are respected. Following some rules, it is possible to replace their global memory usage with message-passing, and parallelize their execution.

TRANSFORMATION RULES

SCOPE

If an handler uses a variable between the reception of two data, then it needs to store it independently of the global memory. The reliance on this memory imposes the handler to not be reentrant. There cannot be several instances of its execution. The stream of incoming data must be processed sequentially.

STREAM

If two handlers causally related rely on the same memory region, the causal relation assures the exclusivity of access. Therefore, the global memory can be replaced by sending the updated memory in the data-flow. As illustrated in figure 4.5, the upstream handler (A1)

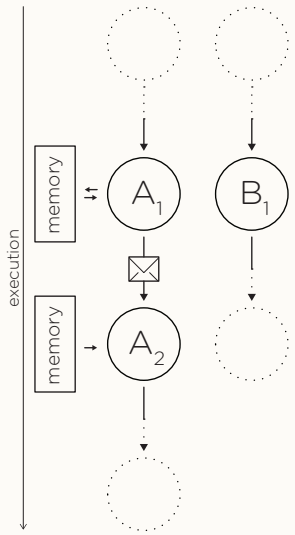


Figure 4.5 – Message passing memory update

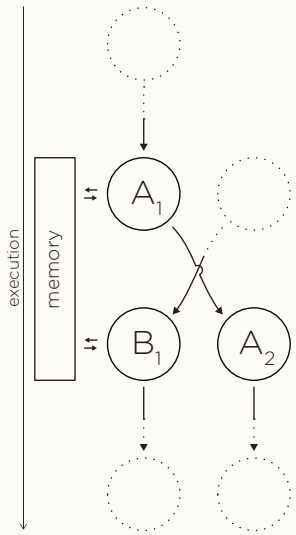


Figure 4.6 – Sequential execution

communicates the memory update to the downstream handler (A2). And each handler has access only to its own memory.

SHARE

However, if the downstream handler modifies this memory, it is not possible to isolate it. The upstream handler cannot be notified in time of this modification. Indeed, the upstream handler might already be processing the next datum in the stream. Moreover, if two handlers not causally related rely on the same memory region, they can access it in any order. They need to be scheduled sequentially to maintain the exclusivity of access, as illustrated in figure 4.6. The handlers (A1) and (B1) rely on the same memory, and are scheduled sequentialy. Whereas the handler (A2) is independent, and can be executed in parallel.

The variables defined before the *start* are available for the whole chain, and require synchronization for concurrent execution of the same chain. Whereas the variables defined inside the chain are available only in this chain, and don't require synchronization, they are sent in the stream.

×

By distributing the global memory following these rules, the sequential scheduling can be loosen while preserving invariance to parallelize the execution. This distribution - hence the parallelization - only depends on the memory dependencies between handlers. Of course, at first, the dependencies will remain tight as the focus is on productivity. But, developers can continuously iterate on implementation to loosen the dependencies and improve efficiency.

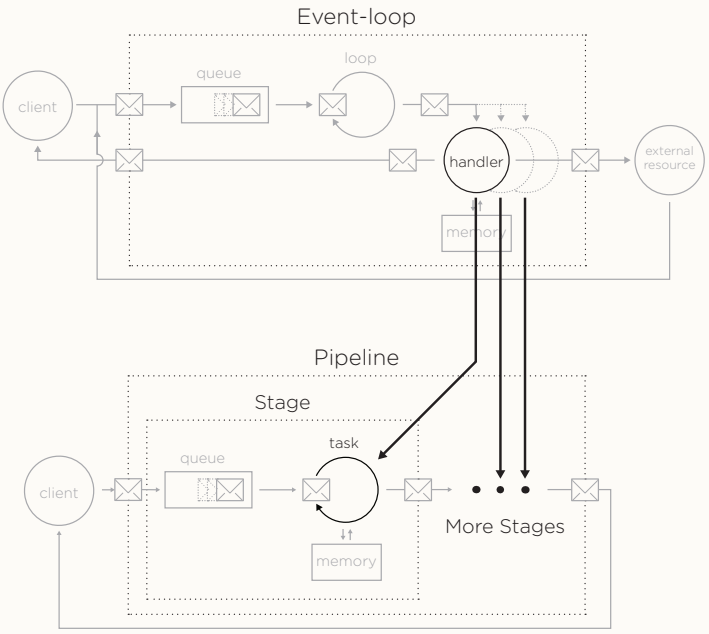


Figure 4.7 – Equivalence between handlers and tasks

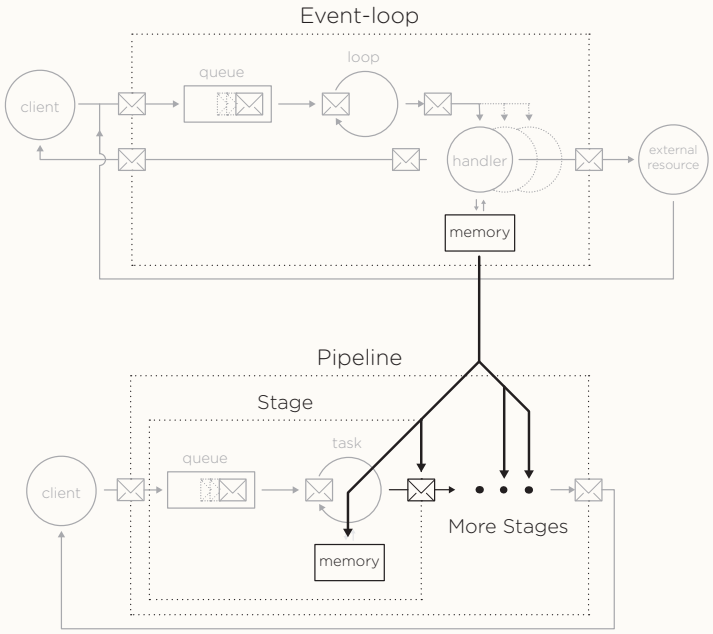


Figure 4.8 – Distribution of the global memory abstraction with message passing

The implementation of the distribution of the global memory is fully addressed in next chapter, in section 5.2.2.

FROM EVENT-LOOP TO PIPELINE

Concretely, this transformation turns handlers from the event-driven execution models into stages of the pipeline, as illustrated in figure 4.7. And it distributes the global memory into these different stages, as illustrated in figure 4.8. The details of these two execution models important for this transformation are presented in the next section.

4.2 EXECUTION MODELS

The event-driven execution model and the pipeline execution model were already briefly presented in chapter 2, section 2.1.2, page 15. The next paragraphs dive into the details of each execution model in regard of the transformation.

4.2.1 EVENT-DRIVEN EXECUTION MODEL

The event-driven execution model processes a queue of asynchronous events by scheduling handlers cooperatively. 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

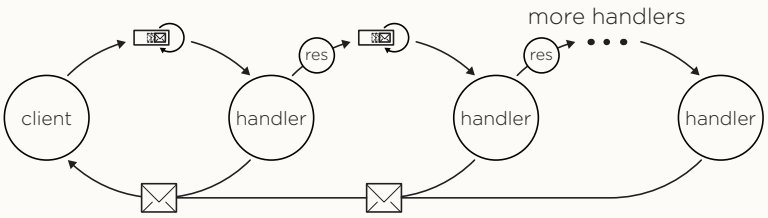


Figure 4.9 – Chain of continuations

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 [177, 83].

CONTINUATION PASSING STYLE

A continuation is a function passed as an argument to a function call. The continuation is invoked after the completion of the callee, to continue the execution. In the event-driven execution model, the continuation is invoked as a new handler, to avoid blocking the caller until its completion. At its invocation, the continuation retrieves both the caller context and the result of the callee through a closure. Listing 4.1 illustrates an example of continuation in *Node.js*.

```
1 callee(input, function continuation(error, result) {
2   if (error)
3     throw error;
4   // ... modify result
5   console.log(result);
6 });
```

Listing 4.1 – Example of a continuation

Asynchronous continuations cannot be composed to chain their execution like synchronous functions, as illustrated in listing 4.2. This nested construction is sometime difficult to follow. Promises improve continuations to allow this composition. They allow to arrange a sequence of asynchronous operations in a chain, similar to a pipeline.

```
1
2 callee(input, function continuation(error, result) {
3   if (error)
4     throw error;
5   // ... modify result
6   nestedCallee(result, function continuation(error, nestedResult) {
7     if (error)
8       throw error;
9     // ... modify result
10    superNestedCallee(nestedResult, function continuation(error,
          superNestedResult) {
11      // ... and so on ...
12    }
13  });
14 });
```

Listing 4.2 – Example of a sequence of continuations

PROMISE

A Promise is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. Listing 4.3 illustrates a simple promise. In Javascript, promises expose a `then` method which expects a continuation to continue the execution with the result of the deferred operation. This method allows to chain Promises one after the other, as illustrated in listing 4.4.

```
var promise = callee(input)

promise.then(function onSuccess(result) {
  console.log(result);
}, function onError(error) {
  throw error;
});
```

Listing 4.3 – Example of a Promise

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

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

Listing 4.4 – 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

The fluxional execution model is inspired by the pipeline architecture. It is the target for the transformation presented in this thesis. It executes programs written in the fluxional language, whose grammar is presented in figure 4.10. It intends to provide scalability to web applications with a granularity of parallelism at the function level.

The functions of an application ⟨program⟩ are encapsulated in autonomous execution containers named *fluxions* ⟨flx⟩.

FLUXION

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, similarly to the actors model. The messaging system queues

⟨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 output messages for the event loop to process them later by calling the downstream *fluxions*.

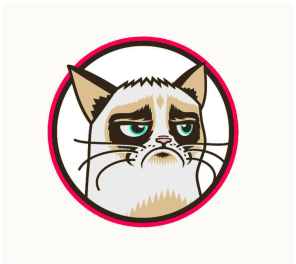
In addition to message passing, the execution model allows *fluxions* to communicate by sharing state between their *contexts*. The fluxions that need this synchronization are grouped with the same tag, and loose their independence.

STREAMS

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. *Start* streams are indicated with a double arrow (`>>`) and *post* streams with a simple arrow (`->`).

4.2.3 EXAMPLES

ILLUSTRATION OF THE FLUXIONAL EXECTION MODEL



Even if the fluxional execution model is not designed to directly develop applications, a first application was developed with it, to test it. This application doesn't use the transformation, nor the continuous development advocated in this thesis. Nonetheless, it is related here to illustrate the basic functioning of the fluxional execution model.

The application is accessible online at `http://grumpy.etnbrd.com`, and the source code is available at `https://github.com/etnbrd/grumpy`. And the network of fluxions structuring the applications can be monitored in realtime at `http://grumpy.`

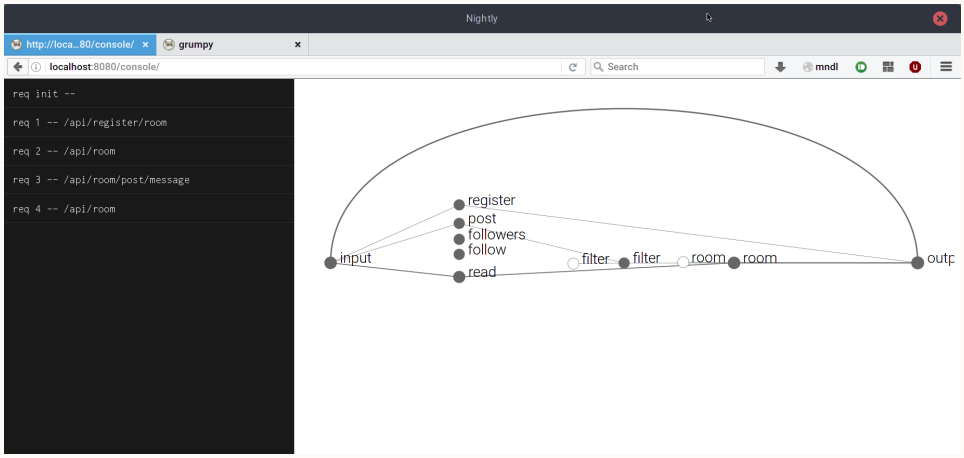


Figure 4.11 – Screenshot from the grumpy console

`etnbrd.com/console`, as illustrated in figure 4.11. The application allows to post anonymous grumbles into chat rooms. Additionally, a room can subscribe to the grumbles from another room. The reader is invited to play with this application and the monitoring console to quickly grasp an understanding of the fluxional execution model [1].

The network of fluxions is organized into five layers of stages, traversed by the stream of request from left to right. The first and last layers are the input and output, connecting the application with the clients. The second layer contains the fluxions receiving and formatting the request, before passing them to the next layer. The fluxion in the third layer is a simple filter before posting grumbles. It is an example of a fluxion modifying a message from the stream. Finally, each room is instantiated as a new fluxion in the fourth layer, storing the received grumbles in its context.

The left panel logs the requests received by the application, and the path of each request can be traced from stage to stage.

The socket descriptor is stuck with the network interface, hence, it cannot be serialized to flow from one fluxion to another. Instead, the first and last fluxions share their memory, and each request flows with an id to associate it with its socket descriptor.

This example application was developed before the implementation of the isolation of fluxions in the fluxional execution model. So it was possible to share their memory by simply sending a memory reference between two fluxions, without grouping them. This explains the direct link between the input and output fluxions.

The example illustrated in the next paragraphs explains in more details the transformation, and the grouping required for fluxion shar-

[1] The reader might want to keep in mind that crude languages might be present on the platform. It is available freely on the internet for anyone to express itself.

ing memory. And it explains the impact of the memory dependencies on parallelism and scalability.

ILLUSTRATION OF THE APPLICATION TRANSFORMATION

The transformation from continuation passing style to the fluxion execution model is now illustrated with a simple web application.

```
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 4.5 – Example web application

The example application in listing 4.5 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 8, interface the application with the clients. Between these two interface functions, there 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 4.6. The transformation process between the source and the fluxional code is explained in chapter 5, in section 5.2.1.

```
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 4.6 – Example application expressed in the high-level fluxional language

The execution is illustrated in figure 4.12. 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 to ⑥ until the queue is empty. This cycle starts again for each new incoming request causing another `start` message.

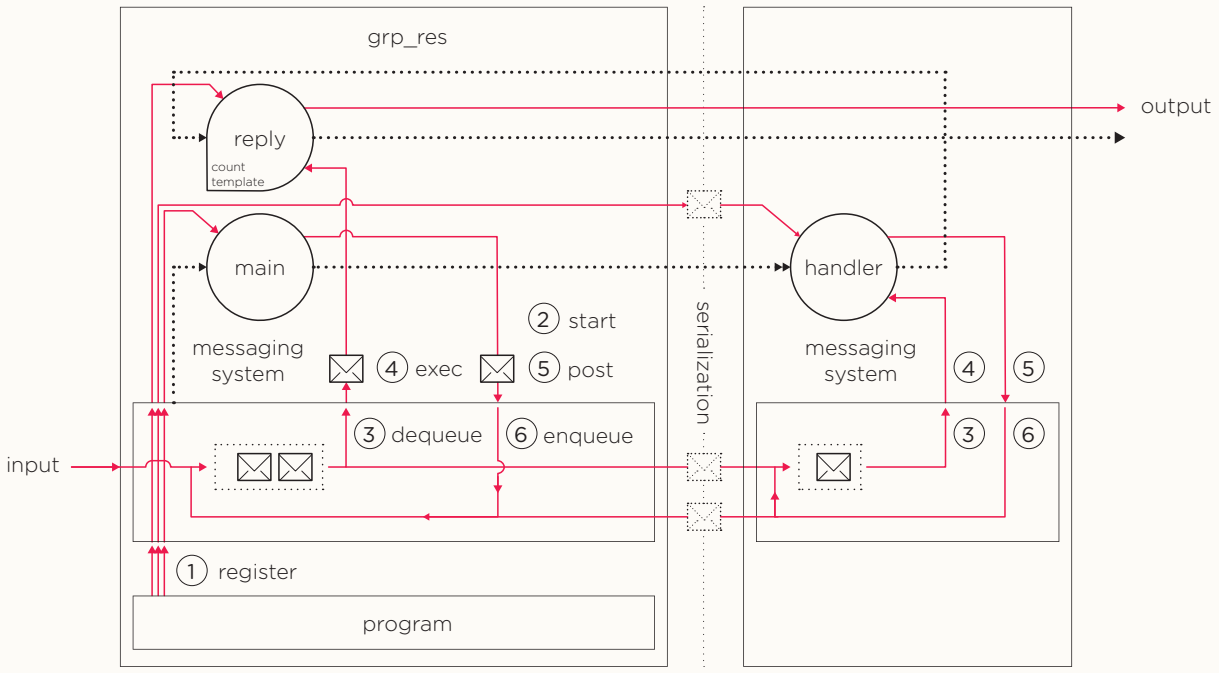


Figure 4.12 – The fluxional execution model in details

The chain of functions from listing 4.5 is expressed in the fluxional language in listing 4.6. 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 variable, 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 avoids the disruptive shifts of technology required by current platforms.

4.3 CONCLUSION

This chapter presented the proposition of this thesis : an equivalence between the event-driven execution model and the pipeline execution model for web applications. The equivalence intends to allows developers to control simultaneously productivity, and efficiency of their implementation. It allows them to continuously have two representations. Developers can then choose their compromize, depending on what they are after. This compromize evolves with the implementation.

The equivalence is based on the event-driven and fluxional execution models presented respectively in section 4.2.1 and 4.2.2. It relies on the similar pipeline organization shared by these two models. It detects the rupture points between the stages to extract the pipeline organization from an event-driven implementation. Then, it isolates these stages so as to allow parallel execution.

We published this work at three occasions. The core idea of these contribution was briefly presented as a poster in December 2014 at Middleware [21], and then more thouroughly presented in April 2015 at Symposium for Application Computing in the track Practical Aspect of Parallel Programming [22]. In the meantime, an intermediate step was presented in April 2015, at the AWeS Workshop at the Eurosys Conference [20]. The implementations presented in these papers are reminded in the next section.

CHAPTER 5

IMPLEMENTATIONS

The transformation allowed by the equivalence from an event-driven program into a distributed network of fluxions is implemented incrementally into two compilers, as presented in figure 5.1. Each compilers is divided into two steps, the identification of the rupture points separating the stages of the pipeline, and the isolation of these stages.

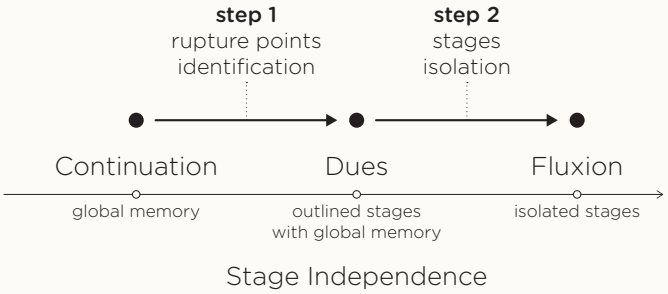


Figure 5.1 – Roadmap

The first compiler focuses on the identification of simple chains of causality between continuations to transform these chains into Promises. However, promises are more expressive than the simple chaining of causal sequentiality. Moreover, 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 identification of chains. To rule out this difference between continuations and Promises, before introducing the first compiler, 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. This second compilers is introduced in section 5.2.

5.1 STEP 1 - DUE COMPILER

5.1.1 DUES

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 convention 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].

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

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

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.

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.

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.

COMPOSITION

Dues arrange the execution flow as a chain of actions to carry on inputs. The composition of Dues in a chain is illustrated in listing 5.3. It is similar to the composition of Promises explained in the previous chapter, section 4.2.1, page 67.

```
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. For example, in listing 5.3 the Due returned by the `then` method line 8 settles when the Due returned by its continuation `my_fn_due_2` settles. It allows to chain continuations one after the other like a pipeline, 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.

EXECUTION ORDER

The transformation of a simple continuation is illustrated in figure 5.2 It applies on 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 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.

$$fn([arguments], continuation) \quad (5.1) \quad \rightarrow \quad fn_{\mathbf{due}}([arguments]).\mathbf{then}(continuation) \quad (5.2)$$

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. To preserve the execution order, the function fn must be asynchronous, or execute the continuation as the last instruction.

EXECUTION LINEARITY

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

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

Figure 5.3 – Composition transformation

The main control flow characteristics, is to allow the execution order to be different from the linearity expressed in the source file. The equivalence must take into account these disruptions when modifying the source.

A call inside a loop yields multiple Dues because of the repetition, while only one can be returned to continue the chain. The others would be discarded. Similarly, a function definition is not executed in situ. It breaks the execution linearity, and prevents a call nested within it to return the Due expected to continue the chain in the parent. Therefore, the composition transformation doesn't apply on chain of Dues nested inside loops or function definitions. It must breaks the chain into simple transformation.

On the other hand, conditional branching leaves the semantic intact in a chain of Dues. If the nested asynchronous function is not called due to branching, the execution of the chain stops as expected. The transformation to a chain of Dues doesn't impact the semantic.

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. The detection of such variables requires to 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 equivalence excludes programs using these statements to keep a lexical scope and be able to infer variable scope statically.

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 online to reproduce the tests at `compiler-due.apps.zone52.org`, and the code is available online at `https://github.com/etnbrd/due-compiler`.

IDENTIFICATION OF CONTINUATIONS

The first compilation step is to identify the continuations and their imbrications. The compiler transforms only *in situ* continuations - these are lambdas. Modifiying continuations that are named functions break the correctness of the application. The modifications operated by the compiler modify the behavior. The modified continuations might be used elsewhere, where the modifications are not expected, hence impact the semantic.

To detect continuations, the compiler looks for callbacks. Not all detected callbacks are continuations, whereas the equivalence is applicable only on the latter. A continuation is a callback invoked only once, asynchronously. This definition is only semantical. 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.

To recognize the two, the compiler would need to have a deep understanding of the semantic of the application. 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 point to its synchronus counter parts. This example is very unlikely, but it shows the limitation.

The compiler identifies callbacks and leaves the identification of compatible continuations among these callbacks to the developer.

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 5.4, the continuation `cont2` is a legitimate child, whereas `cont3` and `cont4` are not because they are siblings, and none have children. The compiler cannot decide to continue the chain with `cont3` or `cont4`, so it leaves them as is.

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

```
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

×

Because Dues are a placeholder for a single outcome, it doesn't allow to express the recurrence of streaming data. Concretely, a Due is created for each datum in the stream. Therefore, continuations doesn't represent the whole streaming pipeline underlying in the application. Listeners are the callbacks starting the streaming pipeline. There is a need for an abstraction for both continuations and listeners.

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

The Due compiler is an intermediary step toward the fluxional compiler presented in section 5.2. This second compiler forms a pipeline with both listeners and continuations, and isolate the stages.

EVALUATION

The Due compiler was evaluated against a set of Javascript projects likely to contain continuations. Because the compilation requires user interaction to detect continuations, the test set was limited to a minimum of about 50 projects to conduct the test in a reasonable time. All the projects in the set were selected from the *Node Package Manager* (*npm*[3]) database to restrict the set to *Node.js* projects [4]. They all depends on the web framework *express*, but not on the most common Promises libraries such as *Bluebird*, *Q* or *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 52 projects. This subset cannot represent the wide possibilities of Javascript, but represents a majority of common cases.

Over the 52 packages the compiler was tested on, 43 packages were incompatible with the compiler and 9 packages were compiled with success.

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 validation assures the compiler to work as expected in most common cases.

Of the 52 projects tested, more than a half, does not contain any compatible continuations. These projects use continuations, but the compiler discards them, because they are not declared *in situ*. The other projects were rejected by the compiler because they contain `with` or `eval` statements. 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. Listeners represent the initiation of stream of data, and are addressed in the next section.

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.

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

[4] At the time of this work, *npm* was still restrained on *Node.js* packages. It is now open to every Javascript packages.

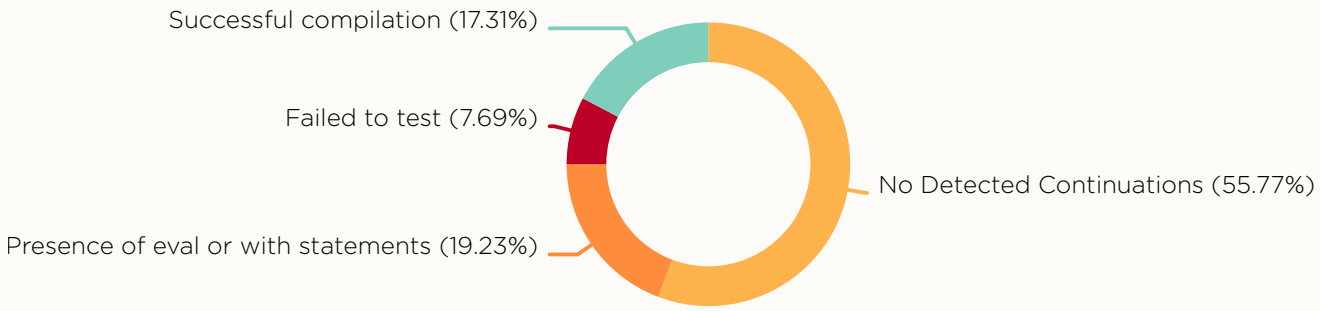


Figure 5.5 – Results of the Due compiler evaluation

On the 52 projects composing the test set

29 (55.77%) do not contain any compatible continuations,

10 (19.23%) are not compilable because they contain `with` or `eval` statements,

4 (7.69%) fail their tests before the compilation, and

9 (17.31%) compile successfully.

5.2 STEP 2 - FLUXIONAL COMPILER

The second contribution of this thesis is the equivalence between a global memory abstraction and a distributed memory. It tackles the problems arising from the replacement of the global memory synchronizations with 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.

Like the previous compiler, this compiler is organized into two main steps. The first step is the identification of the rupture points between fluxions, addressed in section 5.2.1. The second step is the isolation between the fluxions, addressed in section 5.2.2. The compiler is tested on a real-case, to expose its limits in section 5.2.3.

5.2.1 FLUXIONS COMPILER

The source language for this transformation is of higher-order to allow the modularity required for productivity. Moreover, it is implemented as an event-loop to impose the developer to define the causality between asynchronous operations. The compiler transforms a *Node.js* application into a fluxional application compliant with the execution

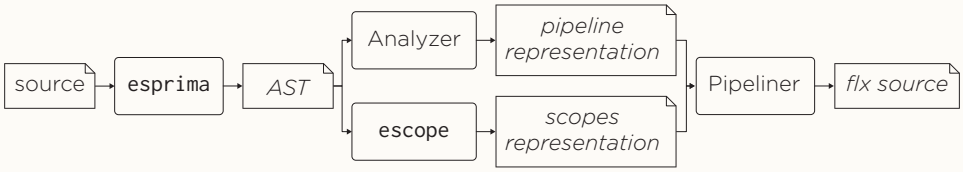


Figure 5.6 – Compilation chain

model described in section 4.2.2. The chain of compilation is described in figure 5.6.

The compiler uses the *estools*[5] suite to parse (`esprima`), analyze (`escope`), manipulate (`estraverse` and `esquery`) and generate (`escodegen`) source code from an Abstract Syntax Tree (AST). It is tailored for – but not limited to – web applications using *Express*[6], the most used *Node.js* web framework. The compiler extracts an AST from the source with `esprima`. From this AST, the *Analyzer* step identifies the rupture points between the different application parts. This first step outputs a pipeline representation of the application. In this 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, explained in section 5.2.2, analyzes the pipeline representation and the scopes representation to distribute the shared memory into independent groups of fluxions.

DETECTION

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.

TODO

Similarly as in the Due compiler, the detection of asynchronous callees is done by the developer. It 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`. The callback functions declared *in situ* are trivially detected in the AST. The compiler discard callbacks not declared

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

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

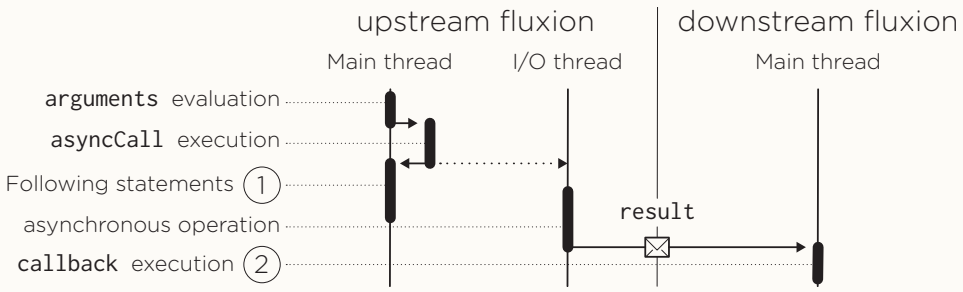


Figure 5.7 – Rupture point interface

in situ, to avoid altering the semantic by moving or modifying their definitions.

5.2.2 FLUXIONS ISOLATION

As a rupture point occurs between an asynchronous caller and a callback defined *in situ*, it eventually breaks the chain of scopes. If the caller and the callback are separated, it breaks the closure of the callback. The callback in the downstream fluxion cannot access the scope of its parent as expected. The pipeliner step replaces the need for this closure, allowing application parts to be isolated, and 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.

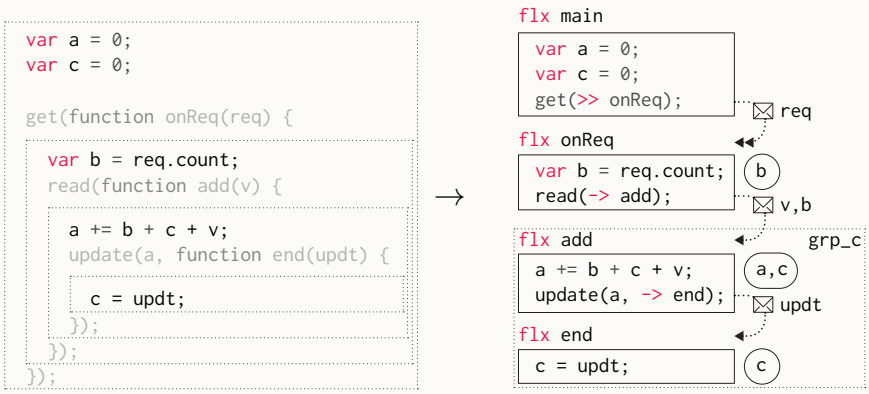


Figure 5.8 – Variable management from Javascript to the high-level fluxional language

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 some 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, in the current *post* chain, the execution of the upstream fluxion is assured to end before the execution of the downstream fluxion, because of their causality. 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. 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

The compiler is tested 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.

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

```
1 var express = require('express'),
2     app = express(),
3     routes = require('gifsockets-middleware'),
4     getRawBody = require('raw-body');
5
6 function bodyParser(limit) {
7   return function saveBody(req, res, next) {
8     getRawBody(req, {
9       expected: req.headers['content-length'],
10      limit: limit
11    }, function (err, buffer) {
12      req.body = buffer;
13      next();
14    });
15  };
16 }
17
18 app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.
      writeTextToImages);
19 app.listen(8000);
```

Listing 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 from the evaluation set of the Due compiler because it is 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`.

COMPILATION

The compilation result is in listing 5.7. The function call `app.post`, line 18, is a rupture point. However, its callbacks, `bodyParser` and `routes.writeTextToImages` are not declared *in situ*. They are evaluated as functions only at runtime. As precised previously, the compiler discards these callbacks to avoid altering the semantic.

```
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 [req, next]);
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. In this situation fluxions are quite similar to Dues regarding memory shareing. Our goal is to isolate the two fluxions, to be able to safely parallelize their executions.

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. `req` is an alias to a memory location used in multiple palces in code. Therefore, 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. The next paragraph details how this situation is handled 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 the aliases of this variable. However, the side effect resulting from this association impacts a variable in the scope of the next callback, `routes.writeTextToImages`. In this test case, the application is modified manually to explicitly propagate this side-effect to the next callback through the function `next`. The modifications of this function are explained further 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.

5.2.4 LIMITATIONS

The static analysis used for this compiler presents some limitations. It is unable to analyze code with dynamic behaviors. Higher-order programming leads to more productivity partly beacuse it rely on such dynamic behavior to extend expressivity. Precisely, it allows more levels of indirections.

LEVELS OF INDIRECTIONS

The indirection is an abstraction between the value, and its manipulation. In listing 5.9, the variables `a` and `b` point both to the same memory object. The function `fn` introduces a level of indirection between the real object `a` and its manipulation handle, `b`;

```
1  var a = {
```

```
2       // an object;
3     };
4
5 fn(b) {
6   // modify b;
7 }
8
9 fn(a);
```

Listing 5.9 – One level of Indirection

UNCERTAINTIES

The indirection is trivial to resolve in listing 5.9. It only needs to have access to the definition of `a` and of `fn`. However, in listing 5.10, the array `handlers` introduces a new level of indirection. The static analysis now needs to have access to the definition of `i` and of the `handlers`. If this definition is provided by an external input, it is not available statically, hence, it adds an uncertainty during the analysis.

```
1 var a = {
2       // an object;
3     },
4     handlers = [
5       // definition of fn handlers;
6     ],
7     i = ?;
8
9 handlers[i](a);
10 handlers[i+1](a);
```

Listing 5.10 – Two levels of indirection

These examples are extremely simplified. A real application contains enough indirections for the static analysis to be overwhelmed by uncertainties, and to be unable to resolve the variables. If a variable is left unresolved, it is impossible to assure its scope and its aliases. Therefore, the compiler is unable to isolate it into a fluxion, or to distribute its modification by messages.

Moreover, it leads the compiler to ignore the rupture points not defined *in situ*, because their modifications could impact the semantic. The reason for this precaution, is that the compiler is unable to assure where the function is used, and the scope of its variables. Therefore, it is unable to assure that the modification will conserve the semantic.

DYNAMIC RESOLUTION

In a web application, this variable `i` might be part of the user request, which is available only at runtime. It eventually introduces an uncertainty.

This dynamic resolution of variables is precisely what increase expressiveness. Trying to resolve them statically is equivalent to restrict expressiveness. No static analysis can overstep these limitations. Only a dynamic analysis could analysis the resolved indirections during run time to overstep these limitations correctly.

CHAPTER 6

CONCLUSION

The web brought a new economic model allowing 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. 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. The platforms at the state of the art propose only compromises between the two. This thesis presented a platform allowing a progressive compromise to fit the economical incentives throughout the evolution of the project.

This chapter summarizes the contributions of this thesis, and evaluates the proposed solution. It finally concludes on the perspectives beyond this thesis.

6.1 SUMMARY

This thesis presented an equivalence between the event-driven execution model and the pipeline execution model. This equivalence was implemented into two compilers. The first compiler allows to identify the rupture point to form chains of stages from programs targeting the event-driven execution model. The resulting chains still depends on a common memory store. The second compiler, stemming from the first one, identifies the entry points of these chains - start rupture points - and enforces isolation to form a parallel pipeline.

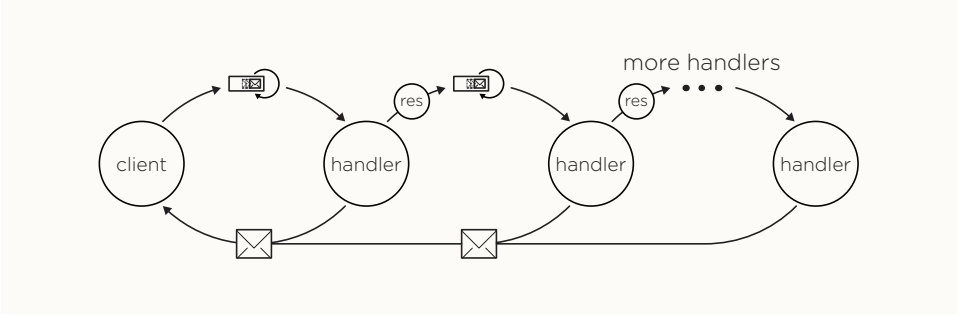
With these two contributions, it is possible to transform the modular representation of an application into a pipeline representation. The modular representation allows development productivity, while the pipeline representation executes efficiently. A development team shall then use these two representations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

The next paragraphs summarizes the two execution models, and the steps of this equivalence from one model to the other.

6.1.1 MODELS

EVENT-DRIVEN EXECUTION MODEL

The event-driven execution model is targeted by productive programming languages. It processes a queue of asynchronous events by scheduling handlers cooperatively. Each handler can asynchronously



request resources and define handlers to receive the resource response. The handlers are chained by these asynchronous requests using continuation passing style. Apart from the resource response, the communication between tow handlers is done via closures. The downstream handler gets access to the environment of its creation through a closure.

They are organized similarly to a pipeline, each handler being a stage, with the data flowing from one stage to the other. However, the handlers still share a common memory store. It allows the higher-order programming required for productivity. But it also avoids the parallelism required for efficiency.

FLUXIONAL EXECUTION MODEL

The fluxional execution model is targeted by a more efficient programming language. It executes an application expressed as a network of independent application parts called fluxions. The fluxions communicate by messages, and form a pipeline similarly to the handlers of the event-driven execution model. They are executed in parallel to distribute the computation across several cores and increase the performance efficiency. They rely on isolated memory store, called context to allow this parallel execution. When several fluxions need to rely on the same memory store, they are grouped, and executed sequentially. When they are independent, they are isolated to increase performance efficiency.

The similarity between the event-driven execution model and the fluxional execution model leads to the equivalence presented in the next paragraph. With this equivalence, the versatility of fluxions allows to progressively adapt the implementation from a productive, single event-loop, toward an efficient pipeline.

6.1.2 EQUIVALENCE

The equivalence describes the transformation from an application targeting the event-driven execution model to execute them in parallel in the fluxional execution model. This transformation involves two steps: the extraction and isolation of the stages to form the pipeline.

STAGE EXTRACTION

The first step is the identification and extraction of the stages. The equivalence identifies rupture points between stages. A rupture point is an asynchronous call without subsequent synchronization with the caller. It indicates a rupture in the synchronous control-flow, and the boundaries between two handlers. The upstream handler is the one calling the asynchronous call, the downstream handler is the callback provided to the asynchronous call.

There are two kinds of rupture points: *start* and *post*. *Start* rupture points directly receives the input stream, and start the execution of the chain of stages for each new datum in the stream. *Post* rupture points indicates a continuity in the chain of stages.

The difficulty in this compilation step is to identify the asynchronous functions indicating the stages. Because of the dynamic behaviors of Javascript, it is impossible to statically detect these functions. The compiler implemented from the equivalence is currently unable to reliably detect them. Instead the compiler rely on the developer to provide a list of asynchronous function names to extract.

STAGE ISOLATION

The second step is the identification of the memory interdependencies between stages. It intends to isolate the stages so they can be executed in parallel. The common memory is replaced by message-passing, following some rules to preserve consistency.

- If a stage needs to hold a variable from one request to the other, this variable is stored in its context.
- If a downstream stage needs to read a variable from an upstream stage, the variable is sent as part of the message communication.
- If two stages needs to share a variable, they are grouped on the same execution node to safely share parts of their context. Their executions are not parallelized to avoid conflicting accesses.

The difficulty in this step is to identify the memory dependencies between stages. The dynamic behaviors of Javascript, it is impossible to statically identify the aliasing in the memory. The compiler implemented from the equivalence is currently unable identify these interdependencies. It relies on manual manipulations to complete the transformation.

×

These difficulties are details in further details in the next section. It then presents some perspectives to overcome these limitations.

6.2 OVERALL EVALUATION

The equivalence presented in chapter 4 is implemented in a the fluxional compiler, presented in section 5.2. This implementation is evaluated against the criteria presented in chapter 3, Productivity, Efficiency and Adoption.

6.2.1 TRADING PRODUCTIVITY FOR EFFICIENCY

The equivalence intends to disrupt as less as possible the way developer build web applications. The goal is to avoid degrading the productivity, hence the adoption, of the proposed platform. Therefore, the productivity is intended to be the same as the original event-driven platform.

However, in the current state, the compiler implementation is unable to operate the transformation without an external help. The static analysis is unable to correctly detect the aliasing of the memory in Javascript. It avoids developers to use Higher-Order Programming, hence impacts composition. This limitation avoids to improve the current trade-off of productivity for efficiency, as illustrated in table **??**. Indeed, to gain efficiency, developers need to commit efforts to indicate the stages of the pipeline, and assure their dependency.

The manual transformation process yields a distributed application, similarly as the other efficient platforms. And the chapter 3 showed that such applications achieve very good performance efficiency. But the productivity limitation remains. It avoids the current implementation to propose a satisfying compromise between productivity and efficiency. So, the current implementation actually trades productivity for efficiency, similarly to many platform in the state of the art. The perspectives to overcome this limitation are addressed later in section 6.3.

6.2.2 ADOPTION

As observed in the chapter 3, trading productivity for efficiency drastically reduces adoption. Because the current implementation presents the same limitation than the efficient platforms, its adoption is not expected to be different.

Yet, both productivity and efficiency are required for the platform to be adopted by new developers as well as large businesses. Only at this condition, will it reinforce the loop between community and industry. So the current implementation is not expected to be widely adopted, as presented in the table 6.1.

Model	Composition	Encapsulation	→ Productivity	Fine-grain level synchronization	Coarse-grain level message passing	→ Efficiency	Community support	Industrial need	→ Adoption
Fluxional Compiler	2	5	2	5	5	5	3	3	3

Table 6.1 – Summary of the proposed solution

The limitation of static analysis avoids the equivalence to be fully implemented to address the problematic. Hence, this evaluation holds only on the implementation, and not on the equivalence.

When saying that *it is a mistake to attempt high concurrency without help from the compiler*, R. von Behren *et al.* [17] implies that the language alone cannot achieve high concurrency. It is necessary to rely on additional tools, such as a compiler to reach the best compromise between productivity and efficiency. The evaluation of this thesis concludes that static analysis is unable to reach this compromise for the current multi-paradigm languages using higher-order programming. Yet, there exist alternatives to static analysis to reach this compromise. The next paragraph presents some interesting perspectives of this work to further address this problematic.

6.3 PERSPECTIVES

As stated previously, static analysis impacts productivity to favor efficiency. Though, an interesting perspective to continue this work is to implement as a just-in-time compiler. Indeed, the dynamic analysis allowed at run time is more prone to overcome the limitation identified with static analysis.

6.3.1 JUST-IN-TIME COMPILATION

Most Javascript interpreters compile some parts of the code at run time to improve performances. During this compilation, the levels of indirections are mostly resolved. The code is translated directly into lower-level instructions.

Implementing the equivalence in a just-in-time (JIT) compiler could leverage this dynamic resolution. It could analyze the scope of variables resolved dynamically, and isolate the stages accordingly.

RUPTURE POINT DETECTION

The asynchronous functions identifying rupture points are not part of Javascript. They are special functions provided by the interpreter. With the compiler communicating with the interpreter at run time, detecting rupture points become trivial. The interpreter notifies the compiler when an asynchronous function is called. The compiler then identifies the rupture point and isolates it to possibly execute it remotely.

DOMINATOR TREE

To debug the memory in dynamic languages like Javascript, one can use a dominator tree. It is a tree generated at run time indicating the parenting relations between memory objects. With such a tree, the analysis of interdependencies between stages becomes trivial. Each stage can be isolated in a fluxion, and deployed accordingly to its dependencies.

CLOSURE SERIALIZATION

Closures are required to allow higher-order programming. But the static compiler is unable to manipulate closures, as illustrated in section 5.2.3. Closures are generated dynamically by the interpreter. With the compiler communicating with the interpreter, the former can manipulates and serialize them at run time. It can then send closures between fluxions, like any other objects. It enables the use of higher-order programming within the fluxional execution model. Hence, it would allow, to some extent, to improve the compromise between productivity and efficiency. Indeed, the developer is free to use the higher-order programming to compose modules, with a global memory abstraction. Yet, the execution could distribute this global memory abstraction according to the detected interdependencies.

DYNAMIC GROUPING

With the dynamic detection of stages and their dependencies, and the manipulation of closures, fluxions can be registered during the execution of the application. To assure they meet their dependencies, the fluxions are deployed according to their groups. Two fluxions belong to

the same group if they need to share access to some variables. Therefore, they need to be deployed on the same event-loop to share their memory.

SAFE-CHECKING

It is required to safe-check that the compiled code is consistent with the remaining execution. As an example, just-in-time compilers check the type to assure that a compiled function remains conform to the input and output types of its call site. Similarly, it is required to check that the deployment of fluxions doesn't cause inconsistencies.

If a fluxion ready to be deployed belongs to two different groups, these two groups needs to be gathered on the same event-loop. If they were previously deployed on two different event-loops, they need to be moved with their context to be on the same event-loop. Moreover, to assure consistency, they need to be moved when receiving the request that triggered the fluxion ready to be deployed. So that when this new fluxion is executed at this message reception it has access to the contexts of the two groups. For this purpose, the compiler put the execution on hold, and sends a control message downstream to order the move of the fluxions. In this example, the message inquired the distributed interpreters to stop execution, pack the fluxions and their contexts, and send them back to another remote interpreter. To assure consistency, the execution resumes only when all the fluxions are gathered in the same event-loop, with access to the whole shared memory.

×

The perspectives described in the previous paragraphs overcome the limitations of the current implementation of the compiler. They describe the further implementation of the equivalence, as if I were to continue this work.

6.3.2 EVALUATION OF THE PERSPECTIVE

This second evaluation admits that the JIT compilation resolves all the indirections in the memory. Then, the fluxional JIT compiler doesn't need to rely on human interaction. Therefore, the expected productivity is the same as the productivity language used as source.

Naturally, the performance efficiency of the implementation is, at first, the same of this productivity language, as the development is focused on productivity. Some development efforts are required to improve the efficiency. The result from the compiler helps the developer find the bottle necks, and reduce the effort for this shift. With the help from the compiler, the effort for this shift is expected to be less than the current required effort. Instead of redesigning the architecture of the application to immediately isolate components, it is possible to modify

them to progressively loosen their dependencies. As illustrated in table 6.2, this envisioned platform is expected to yield both productivity and efficiency, not at the same time, but when they are required the most.

Moreover, during this decomposition and after, developers can still rely on higher-order programming, even between isolated application parts. In the current state of the art, there is no known platform to offer higher-order programming between distributed parts. This possibility is therefore unknown, and could actually yield to an unrivaled compromise between productivity and efficiency.

Following the insight along this thesis, a platform bringing both productivity and efficiency simultaneously would be greatly adopted. But it requires to be observed in real conditions before drawing this conclusion.

Model	Composition	Encapsulation	→ Productivity	Fine-grain level synchronization	Coarse-grain level message passing	→ Efficiency	Community support	Industrial need	→ Adoption
Fluxional Runtime	5	5	5	5	5	5	?	?	?

Table 6.2 – Summary of the perspective

6.3.3 FINAL THOUGHTS

As I studied for this thesis, I progressively saw the world differently. And as a final note in this thesis, I want to share this view.

ECONOMIC CONSIDERATIONS

The IT industry understood that trading efficiency for productivity could reduce development time and cost, and hardware performance could compensate. This thesis intends to bring a reconciliation between two economical concerns in the development of a web application, the efficiency of execution and the productivity of development. I believe that it is time to take into account both productivity and efficiency.

The IT industry has an important impact on the environment with its increasing carbon footprint. As the digitalization permeates into every aspects of our lives, it is of crucial importance to consider this impact. Therefore, it is time to reduce the efficiency to the minimum required. Yet, with the increasing importance of the IT, development cannot be reserved to experts anymore. Productivity cannot be traded back for efficiency.

ACCESSIBLE AND OMNIPRESENT DEVELOPMENT

This epoch feels like developers are the scribes and the monks after the invention of writing. I believe that in the time to come, development will be made available for everybody. And additionally as reading and writing, developing will be a prerequisite to communicate with peers. It will allow to express dynamic behaviors, and not only static ones, as Bret Viktor already envisioned[1].

This shift might come with the increasing importance of machine learning. Indeed, it allows to define complex dynamic behaviors without specifically describing every corner cases. It feels like the composition of general case behavior that can seamlessly meld at corner cases. If machine learning can become parts of our daily means of communication, it will radically change our interactions with peers, and with the world.

Moreover, with the advent of the smart contracts based on block chain technology, and the Internet of things, it is not far-fetched to imagine our everyday world infused with behaviors defined by others. I believe the difference between a person and its environment will start to dissolve. A limited preview can be drawn in our dependence to Internet, smart-phones and other connected objects. But the possibilities are beyond our current imagination.

SCALABILITY

At the light of this thesis, I understand that scalability boils down to the choice of an organization. The chosen organization determines what should be kept local, versus, what should be spread globally. I believe the same problematic applies to many different everyday organizations, such as economical and social organizations.

For example, in economy, it is important for certain markets to spread globally. The different international markets, such as stocks and foreign currency exchange market, are crucial to spread economical informations worldwide. The variations of prices on these markets yields the informations to direct the consumption and production of every product and raw materials for the entire population. It avoids spoiling resources. On the contrary, the uncontrolled variations of this global economy can be destructive at a wide scale, and must somehow be contained. Local citizen currency is an example of such containment. It contains the scope of these variations within a local region.

To concludes this thesis, I yield the following problematic. How to layout the organizations composing our everyday world for it to be efficient. Economically, socially, and in many other aspect of our everyday lives, I believe designing an efficient organization boils down to choosing which piece of information shall be kept locally, or spread globally.

[1] https://vimeo.com/115154289

BIBLIOGRAPHY

[1] 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.

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

[3] Yuichiro Ajima, Takafumi Nose, Kazushige Saga, Naoyuki Shida, and Shinji Sumimoto. "ACPdl". In: *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware - ESPM '15*. New York, New York, USA: ACM Press, Nov. 2015, pp. 11–18. DOI: 10.1145/2832241.2832242.

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

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

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

[7] 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.

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

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

[10] Joe Armstrong. *Programming Erlang*. Pragmatic Programmers, 2007, p. 519. DOI: 10.1017/S0956796809007163.

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

[12] 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.

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

[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] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing Networking Storage and Analysis SC 12* (Nov. 2012), pp. 1–11. DOI: 10.1109/SC.2012.71.

[16] 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.

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

[18] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services". In: *ACM SIGOPS* 37.5 (2003), p. 268. DOI: 10.1145/1165389.945471.

[19] 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).

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

[21] Etienne Brodu, Stéphane Frénot, Fabien Cellier, and Frédéric Oblé. "A compiler providing incremental scalability for web applications". In: *Proceedings of the Posters & Demos Session on - Middleware Posters and Demos '14*. New York, New York, USA: ACM Press, Dec. 2014, pp. 35–36. DOI: 10.1145/2678508.2678526.

[22] Etienne Brodu, Stéphane Frénot, and Frédéric Oblé. "Transforming Javascript Event-Loop Into a Pipeline". In: *Symposium on Applied Computing, track Practical Aspect of Parallel Programming* (Dec. 2015). DOI: 10.1145/2851613.2851745. arXiv: 1512.07067.

[23] 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.

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

[25] T Buddhika and S Pallickara. "NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments". In: *granules.cs.colostate.edu* ().

[26] 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.

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

[28] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications* 21.3 (Aug. 2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[29] 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.

[30] 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.

[31] 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.

[32] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. "Introducing OpenSHMEM". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model - PGAS '10*. New York, New York, USA: ACM Press, Oct. 2010, pp. 1–3. DOI: 10.1145/2020373.2020375.

[33] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*. Vol. 40. 10. New York, New York, USA: ACM Press, Oct. 2005, p. 519. DOI: 10.1145/1094811.1094852.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. "Spidle: a DSL approach to specifying streaming applications". In: *Proceedings of the 2nd international conference on Generative programming and component engineering - GPCE '03* (2003), pp. 1–17.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Frederica Darema. "The SPMD Model: Past , Present and Future". In: *Parallel Computing*. 2001, p. 1. DOI: 10.1007/3-540-45417-9_1.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

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

[53] H. Carter Edwards and Daniel Sunderland. "Kokkos Array performance-portable manycore programming model". In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '12*. New York, New York, USA: ACM Press, Feb. 2012, pp. 1–10. DOI: 10.1145/2141702.2141703.

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

[55] JI Fernández-Villamor. "Microservices-Lightweight Service Descriptions for REST Architectural Style." In: *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence, ICAART 2010* (2010).

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

[57] 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.

[58] 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.

[59] 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.

[60] M Fowler and J Lewis. *Microservices*. 2014.

[61] 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].

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

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

[64] JJ Garrett. “Ajax: A new approach to web applications”. In: (2005).

[65] Tarek El-Ghazawi and Lauren Smith. “UPC: unified parallel C”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC ’06*. New York, New York, USA: ACM Press, Nov. 2006, p. 27. DOI: 10.1145/1188455.1188483.

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

[67] 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.

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

[69] 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.

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

[71] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. "Polly - Polyhedral optimization in LLVM". In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT '11)* (2011), None.

[72] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan a Thekkath, Yuan Yu, and Li Zhuang. "Nectar : Automatic Management of Data and Computation in Datacenters". In: *Technology* (2010), pp. 1–8.

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

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

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

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

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

[78] 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.

[79] Biao Han, Zhongzhi Luan, Danfeng Zhu, Yinan Ren, Ting Chen, Yongjian Wang, and Zhongxin Wu. "An improved staged event driven architecture for master-worker network computing". English. In: *CyberC 2009 - International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* i (Oct. 2009), pp. 184–190. DOI: 10.1109/CYBERC.2009.5342202.

[80] 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.

[81] 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.

[82] 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.

[83] CT Haynes, DP Friedman, and M Wand. "Continuations and coroutines". In: *LFP '84 Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984), pp. 293–298. DOI: 10.1145/800055.802046.

[84] B He, M Yang, Z Guo, R Chen, and B Su. "Comet: batched stream processing for data intensive distributed computing". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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).

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

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

[92] 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.

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

[94] 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.

[95] R D Hornung and J A Keasler. "The RAJA Portability Layer : Overview and Status". In: (2014).

[96] 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.

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

[98] 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.

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

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

[101] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS Operating Systems Review. Vol. 41. No. 3.* (2007), pp. 59–72. DOI: 10.1145/1272996.1273005.

[102] Dongseok Jang and Kwang-Moo Choe. "Points-to analysis for JavaScript". In: *Proceedings of the 2009 ACM symposium on Applied Computing SAC 09* (2009), p. 1930. DOI: 10.1145/1529282.1529711.

[103] 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.

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

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

[106] 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.

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

[108] Hartmut Kaiser. "HPX – A Task Based Programming Model in a Global Address Space". In: *PGAS 2014.* New York, New York, USA: ACM Press, Oct. 2014, pp. 1–11. DOI: 10.1145/2676870.2676883.

[109] Hartmut Kaiser, Thomas Heller, and Daniel Bourgeois. *Higher-level Parallelization for Local and Distributed Asynchronous Task-Based Programming*. 2015.

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

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

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

[113] 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.

[114] 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.

[115] 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.

[116] 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.

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

[118] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. "Stateful bulk processing for incremental analytics". In: *International Conference on Management of Data* (2010), pp. 51–62. DOI: 10.1145/1807128.1807138.

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

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

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

[122] 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.

[123] Christophe Mauras. “Alpha : un langage equationnel pour la conception et la programmation d’architectures paralleles synchrones”. PhD thesis. Jan. 1989.

[124] McCool Michael D. “Structured parallel programming with deterministic patterns”. In: *HotPar ’10, 2nd USENIX Workshop on Hot Topics in Parallelism* (2010), pp. 14–15.

[125] M Migliavacca and D Eyers. “SEEP: scalable and elastic event processing”. In: *Middleware’10 Posters and Demos Track.* (2010).

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

[127] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. “Naiad”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP ’13* (Nov. 2013), pp. 439–455. DOI: 10.1145/2517349.2522738.

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

[129] 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.

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

[131] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. “S4: Distributed stream computing platform”. In: *Proceedings - IEEE International Conference on Data Mining, ICDM*. 2010, pp. 170–177. DOI: 10.1109/ICDMW.2010.172.

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

[133] Robert W. Numrich and John Reid. “Co-array Fortran for parallel programming”. In: *ACM SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. DOI: 10.1145/289918.289920.

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

[135] 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.

[136] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-So-Foreign Language for Data Processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (June 2008), p. 1099. DOI: 10.1145/1376616.1376726.

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

[138] 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.

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

[140] 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).

[141] 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).

[142] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Frédo Durand, Connelly Barnes, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), pp. 519–530. DOI: 10.1145/2491956.2462176.

[143] 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.

[144] K.H. Randall. "Cilk: Efficient Multithreaded Computing". PhD thesis. 1998.

[145] 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.

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

[147] 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.

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

[149] H. Sackman, W. J. Erikson, and E. E. Grant. "Exploratory Experimental Studies Comparing Online and Offline Programming Performance". In: *Communi* 11.1 (1968), pp. 3–11. DOI: 10.1145/362851.362858.

[150] 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.

[151] 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.

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

[153] Herbert A. Simon. "The architecture of complexity". In: *Proceedings of the American Philosophical Societyty* 6.106 (1962), pp. 467–482. DOI: 10.1109/MCS.2007.384127. arXiv: 0205649 [cond-mat].

[154] Elliott Slaughter, Wonchan Lee, Sean Treichler, and Michael Bauer. "Regent : A High-Productivity Programming Language for HPC with Logical Regions". In: *SC* (2015). DOI: 10.1145/2807591.2807629.

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

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

[157] 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.

[158] 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.

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

[160] 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.

[161] 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.

[162] 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.

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

[164] 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.

[165] 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.

[166] William Thies, Michal Karczmarek, and Saman Amarasinghe. “StreamIt: A language for streaming applications”. In: *Compiler Construction* LNCS 2304 (2002), pp. 179–196. DOI: 10.1007/3-540-45937-5.

[167] 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.

[168] 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.

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

[170] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. *GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation*. en. Jan. 2010.

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

[172] 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.

[173] Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff. "Expressing and exploiting concurrency in networked applications with aspen". In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '07* (Mar. 2007), p. 13. DOI: 10.1145/1229428.1229433.

[174] 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.

[175] 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.

[176] 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.

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

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

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

[180] 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.

[181] 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.

[182] 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.

[183] Sunghwan Yoo, Hyojeong Lee, Charles Killian, and Milind Kulkarni. "InContext : Simple Parallelism for Distributed Applications Categories and Subject Descriptors". In: *HPDC*. New York, New York, USA: ACM Press, June 2011, p. 97. DOI: 10.1145/1996130.1996144.

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

[185] 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).

[186] Tomofumi Yuki, Gautam Gupta, Daegon Kim, Tanveer Pathan, and Sanjay Rajopadhye. "AlphaZ: A system for design space exploration in the polyhedral model". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7760 LNCS (2013), pp. 17–31. DOI: 10.1007/978-3-642-37658-0_2.

[187] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, and Ankur Dave. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Apr. 2012), pp. 2–2. DOI: 10.1111/j.1095-8649.2005.00662.x. arXiv: EECS-2011-82.

[188] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. "UPC++: A PGAS Extension for C++". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115.