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

Etienne Brodu

July 16, 2015

Abstract

TODO translate from below when ready

Résumé

Internet étend l'économie à une échelle spatiale et temporelle sans précédent. Il permet à chacun de mettre un service à disposition de milliards d'utilisateurs, en seulement quelques heures. La plupart des grands services actuels ont commencés comme de simples applications créées dans un garage par une poignée de personnes. C'est cette facilité à l'entrée qui a permis jusqu'à maintenant une telle croissance sur le web. Google, Facebook ou Twitter en sont les exemples les plus flagrants. Pendant le développement d'une application, il est important de suivre cette croissance, au risque de se faire rattraper par la concurrence. Le développement est guidé par les besoins en terme de fonctionnalités, afin de vérifier rapidement si le service peut satisfaire l'audience. Des langages tel que Ruby ou Java se sont imposés comme les languages du web, justement parce qu'ils permettent d'intégrer facilement de nouvelles fonctionnalités.

Si une application répond correctement aux besoins, elle atteindra de manière virale un nombre important d'utilisateurs. Son audience peut prendre plusieurs ordres de grandeurs en quelques jours seulement, ou même en quelques heures suivant comment elle est relayée. Une application est dites scalable si elle peut absorber ces augmentations d'audience. Or il est difficile pour une application dont le développement est guidé par les fonctionnalités d'être scalable.

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

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

Mon travail consiste à tenter d'écarter ce risque dans une certaine mesure. Ma thèse se base sur les deux observations suivantes. D'une part, Javascript

est un langage qui a énormément 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 développeur importante, et est l'environnement d'exécution le plus largement déployé. De ce fait, il se place comme le langage principal du web, détrônant Ruby ou Java. D'autre part, l'execution de Javascript a la particularité de ressembler à un pipeline. La boucle événementielle de Javascript est un pipeline qui s'exécute sur un seul cœur pour profiter d'une mémoire globale. On observe le même flux de messages traversant cette boucle événementielle que dans un pipeline.

L'objectif de ma thèse est de permettre à des applications développées en Javascript d'être automatiquement transformées vers un pipeline d'exécutions repartis. Nous construisons un compilateur permettant d'identifier les fonctions de Javascript et de les isoler dans ce que nous appelons des Fluxions. 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 étant indépendantes, elles peuvent être déplacées d'une machine à l'autre. En transformant automatiquement un programme Javascript en Fluxions, on le rend scalable, sans effort.

Contents

Chapter 1

Introduction

TODO 5p

Chapter 2

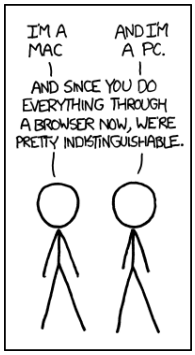
Context and objectives

Contents

2.1 The Web as a platform

2.1.1 From operating systems to the web

With the invention of electronic computing machine, appeared the market for software applications. This market is not limited by marginal production cost ; software being a virtual product, the production and distribution cost for another unit is virtually null. The market is limited by the platform a software can be deployed on. The bigger the platform, the wider the market. There is an economically incentive to standardize and widen the platform, both for the provider, and for the consumer. The first platforms started as products, in competition with other products. Their manufacturers had economical incentive to increase their market share. Microsoft successfully took over the market of operating system in the 90s, and was on the edge of monopoly more than once. But eventually, the product is standardized, and becomes the platform.



Before the internet, this market was limited for distribution by the physical medium. It takes time to burn a CD, or a floppy, and to bring it to the consummer's home. Sir Tim Berners Lee invented the world wide web in 1989. It was initially intended to share scientific documents and results. And it eventually became the distribution medium of choice for every virtual products, software included. It pushed the scalability of software distribution.

Similarly to operating systems, Web browsers started as software products. They exposed innovative features to try to increase their market share. Among others is the ability to run scripts. It allows to deploy and run software at unprecedented scales. The web became the platform. Now, with web services, or Software as a Service (SaaS), the distribution medium of software is so transparent that owning a software product to have an easier access is no longer relevant. We explore now the different languages to write and deploy applications on the web.

2.1.2 The languages of the web

In the early 90's, during the web early development, most of the now popular programming languages were released. Python(1991), Ruby(1993), Java(1994), PHP(1995) and Javascript(1995). With Moore's law predicting exponential

increase in hardware performance, the industry realized that development time is more expensive than hardware. Low-level languages were replaced by higher-level language, trading performance for accessibility. The economical gain in development time compensated the worsen performances of these languages.

Java, developed by Sun Microsystems, imposes itself early as a language of choice and never really decreased. The language is executed on a virtual machine, allowing to write an application once, and to deploy it on heterogeneous machines. The software industry quickly adopted it as its main development language. It is currently the second most cited language on StackOverflow, and used on Github. And is in the first place of many language popularity indexes. However, the software industry wants stable and safe solutions. This prudence generally slows down Java evolution. The language struggled to keep up with the latest trends in software development.

Python is the second best language for everything. It is a general purpose language, currently popular for data science. In 2003, the release of the Django web framworks brought the language to the web development scene.

Ruby was confined in Japan and almost unknown to the world until the release of Rails in 2005. With the release of this web framework, Ruby took-off and is still in active use. It meets the latest trends in software development. And it might had replaced Java if the latter had not been so well adopted in the software industry.

PHP stands for Personal Home Page Tools. It was initially designed to build personal web pages. It might be one of the easiest language to start web development. However, according to several language popularity indexes, it is on a slow decline since a few years. It is generally unfit to grow projects to industrial size.

Since a few years, Javascript is slowly becoming the main language for web development. It is the only choice in the browser. Because of this unavoidable position, it became fast (V8, ASM.js) and usable (ES6, ES7). And since 2009, it is present on the server as well with Node.js This omnipresence became an advantage. It allows to develop and maintain the whole application with the same language. I argue in this thesis, that Javascript is the language of choice to bring a prototype to industrial standards.

2.1.3 Explosion of Javascript popularity

2.1.3.1 In the beginning

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

When Javascript was released in 1995, the world wide web was on the rise.[1] Browsers were emerging, and started a battle to show off the best features and user experience to attract the wider public.[2] Javascript was released to be one of these features on Netscape navigator. Microsoft released their browser Internet Explorer 3 in June 1996 with a concurrent implementation of Javascript. At the time, because of the differences between the two implementations, web pages had to be designed for a specific browser. This competition was fragmenting the web.

Netscape submitted Javascript to Ecma International for standardization in November 1996 to stop this fragmentation. In June 1997, ECMA International released ECMA-262, the first specification of ECMAScript, the standard for Javascript. A standard to which all browser should refer for their implementations.

The initial release of Javascript was designed in a rush. The version released in 1995 was finished within 10 days. And, it was intended to be simple enough to attract unexperienced developers. For these reasons, the language was considered poorly designed and unattractive by the developer community.

But things evolved drastically since. The success of Javascript is due to many factors ; maybe the most important of all is the *View Source* menu that reveals the complete source code of any web application. *The view source menu is the ultimate form of open source*[3]. It is the vector of the quick dissemination of source code to the community, which picks, emphasizes and reproduces the best techniques. It brought open source and collaborative development to the web. Moreover, all web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime in history [1].

When such a language is distributed freely with the tools to reproduce

[1] `http://www.internetlivestats.com/internet-users/`

[2] to get an idea of the web in 1997 : `http://1x-upon.com/`

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

and experiment on every piece of code. And its distribution is carried during the expansion of the largest communication network in history. Then an entire generation seizes this opportunity to incrementally build and share the best tools they can. This collaboration is the reason for the popularity of Javascript on the Web.

2.1.3.2 Rising of the unpopular language

Why does Javascript suck?[4]
Is Javascript here to stay?[5]
Why Javascript Is Doomed.[6]
Why JavaScript Makes Bad Developers.[7]
JavaScript: The World's Most Misunderstood Programming Language[8]
Why Javascript Still Sucks[9]
10 things we hate about JavaScript[10]
Why do so many people seem to hate Javascript?[11]

Javascript started as a programming language to implement short interactions on web pages. The best usage example was to validate some forms on the client before sending the request to the server. This situation hugely improved since the beginning of the language. Nowadays, there is a lot of web-based application replacing desktop applications, like mail client, word processor, music player, graphics editor...

ECMA International released several version in the few years following the creation of Javascript. The first and second version, released in 1997 and 1998, brought minor revisions to the initial draft. The third version, released in the late 1999, contributed to give Javascript a more complete and solid base as a programming language. From this point on, the consideration for Javascript kept improving.

In 2005, James Jesse Garrett released *Ajax: A New Approach to Web Applications*, a white paper coining the term Ajax [2]. This paper points

[4] http://whydoesitsuck.com/why-does-javascript-suck/
[5] http://www.javaworld.com/article/2077224/learn-java/is-javascript-here-to-stay-.html
[6] http://simpleprogrammer.com/2013/05/06/why-javascript-is-doomed/
[7] https://thorprojects.com/blog/Lists/Posts/Post.aspx?ID=1646
[8] http://www.crockford.com/javascript/javascript.html
[9] http://www.boronine.com/2012/12/14/Why-JavaScript-Still-Sucks/
[10] http://www.infoworld.com/article/2606605/javascript/146732-10-things-we-hate-about-JavaScript.html
[11] https://www.quora.com/Why-do-so-many-people-seem-to-hate-JavaScript

the trend in using this technique, and explain the consequences on user experience. Ajax stands for Asynchronous Javascript And XML. It consists of using Javascript to dynamically request and refresh the content inside a web page. It has the advantage to avoid requesting a full page from the server. Javascript is not anymore confined to the realm of small user interactions on a terminal. It can be proactive and responsible for a bigger part in the whole system spanning from the server to the client. Indeed, this ability to react instantly to the user gave to developer the feature to develop richer applications inside the browser. At the time, the first web applications to use Ajax were Gmail, and Google maps[12].

The third version of ECMAScript had been released, and it was homogeneously supported in the browsers. However, the DOM, and the `XMLHttpRequest` method, two components on which AJAX relies, still present heterogeneous interfaces among browsers. Around this time, the Javascript community started to emerge. Javascript framework were released with the goal to straighten the differences between browsers implementations. Prototype[13] and DOJO[14] are early famous examples, and later jQuery[15] and underscore[16]. These frameworks are responsible in great part to the large success of Javascript and of the web technologies.

In the meantime, in 2004, the Web Hypertext Application Technology Working Group[17] was formed to work on the fifth version of the HTML standard. This new version provide new capabilities to web browsers, and a better integration with the native environment. It features geolocation, file API, web storage, canvas drawing element, audio and video capabilities, drag and drop, browser history manipulation, and many mores. It gave Javascript the missing interfaces to become the environment required to develop rich application in the browser. The first public draft of HTML 5 was released in 2008, and the fifth version of ECMAScript was released in 2009. These two releases, ECMAScript 5 and HTML5, represent a mile-stone in the development of web-based applications. Javascript became the programming language of this rising application platform.

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

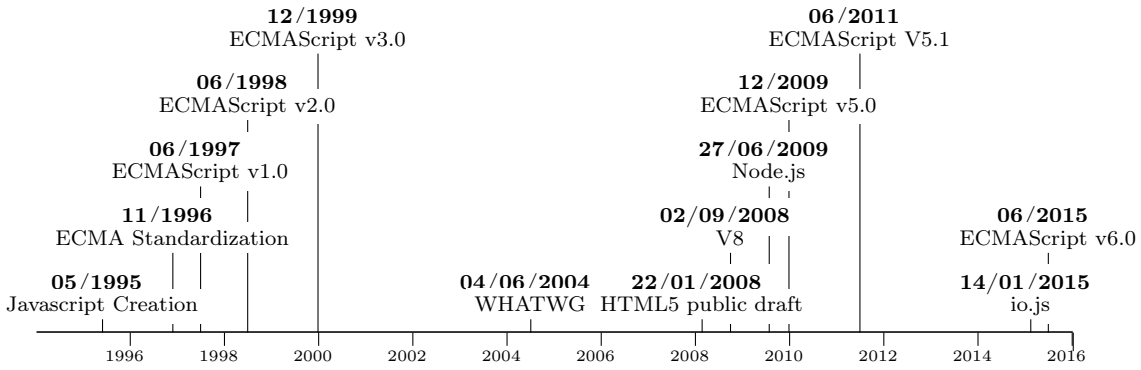
[13] `http://prototypejs.org/`

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

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

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

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



In 2008* Google released V8 for its browser Chrome. It is a Javascript interpreter improving drastically the execution performance with a just-in-time compiler. This increase in performance allowed to push Javascript to the server, with the release of Node.js in 2009. Javascript was initially proposed to develop user interfaces. It is implemented around an event-based paradigm to react to concurrent user interactions. Because of the performance increase, this event-based paradigm proved to be also very efficient to react to concurrent requests of a web server. I will present this event-based paradigm in the next section.

⚠ TODO verify date

2.1.3.3 Current situation

"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

The rise of Javascript is obvious on the web and particularly the open source communities. It also seems to be rising in the software industry. It is difficult to give an accurate representation of the situation because the software industry is opaque for economical reason. In the following paragraphs, I report some indexes that represent the situation globally, both in the open source community and in the more opaque software industry.

Available resources The TIOBE Programming Community index is a monthly indicator of the popularity of programming languages. Javascript ranks 6th on this index, as of April 2015, and it was the most rising language in 2014. It uses the number of results on many search engines about a certain language. The results contains the resources and traces of the activity around the language that are used as a measure of the popularity of a programming language. However, the number of pages doesn't represent the number of readers. The measure used by the TIOBE is controversial, and might not be representative.

Alternatively, the PYPL index is based on Google trends to measure the number of requests on a programming language. Javascript ranks 7th on this index, as of May 2015. This index seems to be more accurate, as it depicts the actual interest of the community for a language. However, it is not representative as it only takes Google search into account.

From these indexes, the major programming languages are Java, C/C++ and C#. The three languages are still the most widely taught, and used to write softwares.

Developers collaboration platforms An indicator of the popularity and usage of a language is the number of developers and projects using it.

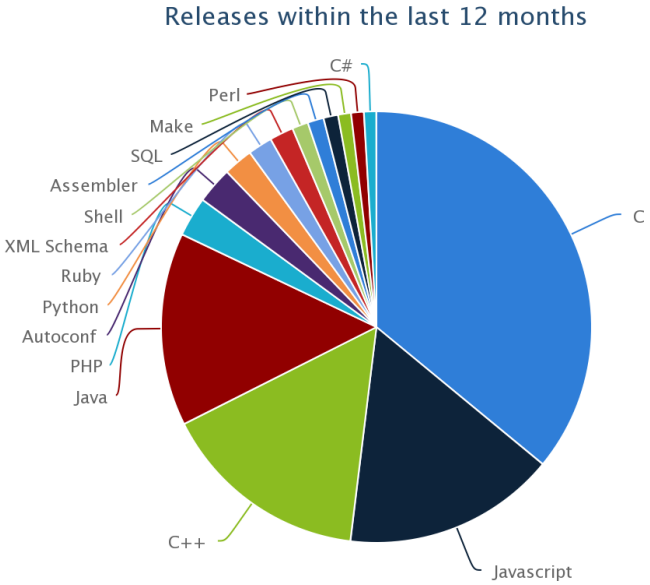
Github is the most important collaborative development platform, with around 9 millions users. Javascript is the most used language on github since mid-2011, with more than 320 000 repositories. The second language is Java with more than 220 000 repositories.

*

TODO : graph of Github repositories by languages

StackOverflow is the most important Q&A platform for developers. It is a good representation of the activity around a language. Javascript is the language the most cited on StackOverflow, with more than 890 000 questions. The second is Java with around 880 000 questions.

Black Duck Software helps companies streamline, safeguard, and manage their use of open source. For its activity, it analyzes 1 million repositories over various forges, and collaborative platforms to produce an index of the usage of programming language in open source communities. Javascript ranks second. C is first, C++ third and Java fourth. These four languages represent about 80% of all programming language usage in open source communities.



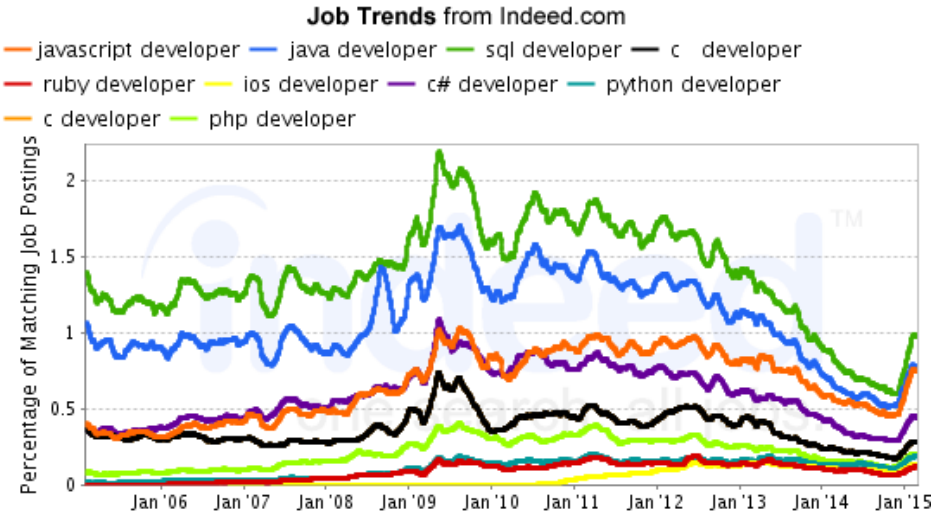
*

Black Duck

⚠

TODO redo this graph, it is ugly.

Jobs The software industry is rather closed sourced, and its activity is rather opaque. All these previous metrics are representing the visible activity about programming language, but are not representative of the software industry. The trends on job openings gives an hint of the direction the software industry is heading towards. The job searching platform *Indeed* provides some trends over its database of job propositions. Javascript developers ranked at the third position, right after SQL and Java developers. Over the 5 last years, the number of job position for Javascript developers increased so as to almost close the gap with Java. This position indicate that Javascript is increasingly adopted in the software industry.



*

TODO redo this graph, it is ugly.

All these metrics represent different faces of the current situation of the Javascript adoption in the developer community. With the evolution of web applications development and increased interest in this domain, Javascript is assuredly one of most important language of this decade. It is widely used in open source projects, and everywhere on the web, as well as in the software industry.

2.2 Highly concurrent web servers

*

This section needs review

2.2.1 Concurrency

The Internet allows interconnection at an unprecedented scale. There is currently more than 16 billions devices connected to the internet, and it is growing exponentially[18]. This massively interconnected network gives the ability for a web applications to be reached at the largest scale. A large web application like google search receives about 40 000 requests per seconds. That is about 3.5 billions requests a day[19]. Such a web application needs to be highly concurrent to manage a large number of simultaneous request. Concurrency is the ability for an application to make progress on several

[18] http://blogs.cisco.com/news/cisco-connections-counter
[19] http://www.internetlivestats.com/google-search-statistics/

tasks at the same time. For example to respond to several simultaneous requests, a task is a part in the response to a request. *

TOOD define more clearly what is a task

In the 2000s, the limit to reach was to process 10 thousands simultaneous connections with a single commodity machine[20]. Nowadays, in the 2010s, the limit is set at 10 millions simultaneous connections at roughly the same price[21]. With the growing number of connected devices on the internet, concurrency is a very important property in the design of web applications.

2.2.1.1 Scalability

The traffic of a popular web application such as Google search is huge, and it remains roughly stable because of this popularity. There is no apparent spikes in the traffic, because of the importance of the average traffic. However, the traffic of a less popular web application is much more uncertain. If the web application fits the market need, it might become viral when it is efficiently relayed in the media. For example, when a web application appears in the evening news, it expects a huge spike in traffic. With the growth of audience, the number of simultaneous requests obviously increases, and the load of the web application on the available resources increases as well. The available resources needs to increase to meet the load. This growth might be steady and predictable enough to plan the increase of resources ahead of time, or it might be erratic and challenging. The spikes of a less popular web application are unpredictable. Therefore, the concurrency needs to be expressed in a scalable fashion. An application is scalable, if the growth of its audience is proportional to the increase of its load on the resources. For example, if a scalable application uses one resource to handle n simultaneous requests, it will use k resource to handle two times n simultaneous requests. With k being constant, for n ranging from tens to millions of simultaneous requests. Scalability assures that the resource usage is not increasing exponentially in function of the audience increase ; it increases roughly linearly.

2.2.1.2 Time-slicing and parallelism

Concurrency can be achieved on hardware with either a single or several processing units. On a single processing unit, the tasks are executed sequentially ; their executions are interleaved in time. On several processing unit,

[20] `http://www.kegel.com/c10k.html`
[21] `http://c10m.robertgraham.com/p/manifesto.html`

the tasks are executed in parallel. Parallel executions reduce computing time over sequential execution, as it uses more processing units.

If the tasks are completely independent, they can be executed in parallel as well as sequentially. This parallelism is a form of scalable concurrency, as it allows to stretch the computation on available hardware to meet the required performance, at the required cost. This parallelism is used in operating system to execute several applications concurrently to allow multi-tasking.

However, the tasks of an application are rarely independent. The tasks need to coordinate their dependencies to modify the global state of the application. This coordination limits the possible parallelism between the tasks, and might impose to execute them sequentially. The type of possible concurrency, sequential or parallel, is defined by the required coordination between the tasks. Either the tasks are independents and they can be executed in parallel, or the tasks need to coordinate a common state, and they need to be executed sequentially to avoid conflicting accesses to the state.

2.2.2 Interdependencies

It is easier to understand the possible parallelism of a cooking recipe than an application. That is because the modifications to the state are trivial in the cooking recipe, hence the interdependencies between operations. It is easy to understand that preheating the oven is independent from whipping up egg whites. While the interdependencies are not immediately obvious in an application. *

TODO is this metaphor useful here ? if yes, continue to a transition

2.2.2.1 State coordination

The interdependencies between the tasks impose the coordination of the global application state. This coordination happen either by sending events from one task to another, or by modifying a shared memory.

If the tasks are independent enough, they never need access to a state at the same time. The coordination of the state of the application can be done with message passing. They pass the states from one task to another so as to always have an exclusive access on the state. As example, applications built around a pipeline architecture define independent tasks arranged to be executed one after the other. The tasks pass the result of their computation to the next. These tasks never share a state.

If the tasks need concurrent accesses to a state, they cannot efficiently pass the state from one to the other repeatedly. They need to share and to coordinate their accesses to this state. Each access needs to be exclusive to avoid corruption. This exclusivity is assured differently depending on the scheduling strategy.

2.2.2.2 Task scheduling

There is roughly two main scheduling strategy to execute tasks sequentially on a single processing unit : preemptive scheduling and cooperative scheduling. The state coordination presented previously is highly depending on the scheduling strategy.

Preemptive scheduling is used in most execution environment in conjunction with multi-threading. The scheduler allows each task to execute for a limited time before preempting it to let another task execute. It is a fair and pessimistic scheduling, as it grant the same amount of computing time to each task. However, as the preemption might happen at any point in the execution, it is important for the developer to lock the shared state before access, so as to assure exclusivity. This protection is known to be hard to manage.

In cooperative scheduling, the scheduler allows a task to run until the task yield the execution back. Each task is an atomic execution ; it is never be preempted, and have an exclusive access on the memory. It gives back to the developer the control over the preemption. It seems to be the easiest way for developers to write concurrent programs efficiently. Indeed, I presented in the previous section the popularity of Javascript, which is often implemented on top of this scheduling strategy (DOM, Node.js).

As I explained the different paradigms for writing concurrent program in this subsection, it appears that the main problem is to assure to the developer for each task the exclusive access to the state of its application. This assurance is called invariance.

2.2.2.3 Invariance

I call invariance the assurance given that the state accessible from a task will remain unchanged during its access to avoid corruption, and more generally to allow the developer to perform atomic modifications on the state. This assurance allows the developer to regroup operations logically so as to

perform all the operations without interference from concurrent executions. The same concept is found in transactional memory.

In a multi-process application, there is no risk of corrupted state by simultaneous, conflicting accesses. The invariance is made explicit by the developer as the memory needs to be isolated inside each process. The invariance is assure at any point in time because the process remain isolated.

In a cooperative scheduling application, the developer is aware of the points in the code where the scheduler switches from one concurrent execution to the other, so it can manage its state in atomic modification. The invariance is assured, because any region in the memory can be accessed only by one task at a time.

Between these two invariances, the locking mechanisms seems to be a promising compromise. The developer defines only the shared states, and these are locked only when needed. However, it increases the complexity of the possible locked combination, leading to unpredictable situations, such as deadlock, and so on. The locking mechanisms are known to be difficult to manage, and sub-optimal. Indeed, they are eventually as efficient as a queue to share resources.

For the rest of this thesis, I focus only on the invariances provided by the multi-process paradigm and the cooperative scheduling. They are similar, because the developer defines sequence of instructions with atomic access to the memory. And in both paradigms, these sequences communicate by sending messages to each other. The difference is that in the multi-process paradigm, the developer defines the region and the isolated memory, while in the cooperative scheduling, the developer defines only the region, and the memory is isolated by the exclusivity in the execution.

This difference seems to be crucial in the adoption of the technology by the developer community. As we will see in the next subsection, the parallelism of multi-process is difficult to develop, but provide good performances, while the sequentiality of the cooperative scheduling is easier to develop, but provide poor performances compared to parallelism.

*

⚠

TODO this paragraph needs review

2.2.3 Technological shift

2.2.3.1 Scalable concurrency

Around 2004, the so-called Power Wall was reached. The clock of CPU is stuck at 3GHz because of the inability to dissipate the heat generated at higher frequencies. Additionally, the instruction-level parallelism is limited. Because of these limitations, a processor is limited in the number of instruction per second it can execute. Therefore, a coarser level of parallelism, like the task-level, multi-processes parallelism previously presented is the only option to achieve high concurrency and scalability. But as I presented previously, this parallelism requires the isolation of the memory of each independent task. This isolation is in contradiction with the best practices of software development, hence, is difficult to develop for common developers. It creates a rupture between performance and development accessibility.

2.2.3.2 The case for global memory

The best practices in software development advocate to design a software into isolated modules. This modularity allows to understand each module by itself, without an understanding of the whole application. The understanding of the whole application emerges from the interconnections between the different modules. A developer need only to understand a few modules to contribute to an application of hundreds or thousands of modules.

Modularity advocates three principles : encapsulation, a module contains the data, as well as the functions to manipulate this data ; separation of concerns, each module should have a clear scope of action, and this scope should not overlap with the scope of other modules ; and loose coupling, each module should require no, or as little knowledge as possible about the definition of other modules. The main goal followed by these principles, is to help the developer to develop and maintain a large code-base.

Modularity is intended to avoid a different problem than the isolation required by parallelism. The former intends to avoid unintelligible spaghetti code ; while the latter avoids conflicting memory accesses resulting in corrupted state. The two goals are overlapping in the design of the application. * Therefore, every language needs to provide a compromise between these two goals, and specialized in specific type of applications. I argue that the more accessible, hence popular programming languages choose to provide modularity over isolation. They provide a global memory at the sacrifice

⚠ TODO needs more explanations -> so it is hard for dev to do both ? Why exactly ?

of the performance provided by parallelism. On the other hand, the more efficient languages sacrifice the readability and maintainability, to provide a model closer to parallelism, to allow better performances. * *

⚠ TODO instead of language, use a more generic term to refer to language or infrastructure

2.2.3.3 Rupture

Between the early development, and the maturation of a web application, the development needs are radically different. In its early development, a web application needs to quickly iterate over feedback from its users. *"Release early, release often"*, and *"Fail fast"* are the punchlines of the web entrepreneurial community. The development team quickly releases a Minimum Viable Product as to get these feedbacks. The development reactivity is crucial. The first reason of startup failures is the lack of market need[22]. Therefore, the development team opt for a popular, and accessible language.

⚠ TODO justification and examples. What are modular application, or parallel applications ?

As the application matures and its audience grows, the focus shift from the development speed to the scalability of the application. The development team shift from a modular language, to a language providing parallelism.

This shift brings two problems. First, the development team needs to take a risk to be able to grow the application. This risk usually implies for the development team to rewrite the code base to adapt it to a completely different paradigm, with imposed interfaces. It is hard for the development team to find the time, hence the money, or the competences to deploy this new paradigm. Indeed, the number two and three reasons for startup failures are running out of cash, and missing the right competences. Second, after this shift the development pace is different. Parallel languages are incompatible with the commonly learned design principles. The development team cannot react as quickly to user feedbacks as with the first paradigm.

This technological rupture proves that there is economically a need for a a more sustainable solution to follow the evolution of a web application. A paradigm that it is easy to develop with, as needed in the beginning of a web application development, and yet scalable, so as to be highly concurrent when the application matures.

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

2.3 Equivalence

I argue that the language should propose to the developer an abstraction to encourage the best practices of software development. Then a compiler, or the execution engine, can adapt this abstraction so as to leverage parallel architectures. So as to provide to the developer a usable, yet efficient compromise. We propose to find an equivalence between the invariance proposed by the cooperative scheduling paradigm and the invariance proposed by the multi-processes paradigm in the case of web applications.

2.3.1 Architecture of web applications

2.3.1.1 Real-time streaming web services

*

⚠

The need for invariance in the streaming applications : it can be emulated by message passing. Indeed the data flows from one processing step to the other, with few retro-progation of state (don't mention retro-propagation yet)

This equivalence intends not to be universal. It focuses on a precise class of applications : web applications processing stream of requests from users in soft real-time.

Such applications are organized in sequences of concurrent tasks to modify the input stream of requests to produce the output stream of responses. This stream of data stand out from the pure state of the application. The data flows in a communication channel between different concurrent tasks, and is never stored on any task. The state represents a communication channel between different instant in time, it remains in the memory to impact the future behaviors of the application. The state might be shared by several tasks of the application, and result in the needs for coordination presented in the previous section. In this thesis I study two programming paradigm derived directly form the cooperative scheduling and the multi-process paradigms presented in previous sections to be applied in the case of real-time web applications. The event-loop execution engine is a direct application of the cooperative scheduling, and the pipeline architecture is a direct application of the multi-process paradigm.

2.3.1.2 Event-loop

The event-loop is an execution model using asynchronous communication and cooperative scheduling to allow efficent execution of concurrent tasks on a single processing unit. It relies on a queue of event, and a loop to process each event one after the other. The communications are asynchronous to let the

application use the processor instead of waiting for a slow response. When the response of a communication is available, it queues an event. This event is composed of the result of the communication, and of a function previously defined at the communication initiation, to continue the execution with the result. In the Javascript even-loop, this function is defined following the continuation passing style, and is named a callback. After processing the result, this callback can initiate communications, resulting in the queuing of more events.

In this model, the data is the result of every communication operations - starting with the received user request - flowing through a sequence of callbacks, one after the other. The state contains all the variables remaining in memory from one request to the other, and from one callback to the other. In Javascript, it includes the closures.

*

⚠ TODO schema of an event-loop

2.3.1.3 Pipeline

The pipeline software architecture uses the multi-process paradigm and message passing to leverage the parallelism of a multi-core hardware architectures for streaming application. It consists of many processes treating and carrying the flow of data from stage to stage. This flow of data consist roughly of the requests, and associated data from the user, as well as the necessary state coordination between the stages. Each stage has its independent memory to hold its own state from one request to another.

*

*

⚠ TODO it is not universal, but multi-process paradigms are also oriented around event-loops. An Event-loop is a multi-process on one machine. A multi-process is multiple event-loop running different part of the same program.

The pipeline architecture and the event-loop model present similar execution model. Both paradigms encapsulate the execution, in callbacks or processes. Those containers are assured to have an exclusive access to the memory. However, they provide two different memory models to provide this exclusivity. It results in two distinct ways for the developer to assure the invariance, and to manage the global state of the application. The event-loop shares the memory globally through the application, allowing the best practice of software development. It is possibly the reason of the wide adoption of this programming model by the community of developers.

I argue in this thesis that it is possible to provide an equivalence between the two memory models for streaming web application. In the next subsection, I present the similarity in the execution model, and the differences in

⚠ TODO schema of a pipeline

the memory model for which an equivalence is necessary. Such equivalence would allow to transform an application following the event-loop model to be compatible with the pipeline architecture. This transformation would allow the development of an application following a programming model allowing the best practices of software development, while leverageing the parallelism of multi-core hardware architecture.

2.3.2 Equivalence

2.3.2.1 Rupture point

The execution of the pipeline architecture is well delimited in isolated stages. Each stage has its own thread of execution, and is independent of the others. On the other hand, the execution of the event-loop seems pretty linear to the developer. The continuation passing style nest callbacks linearly inside each others. The message passing linking the callbacks is transparently handled by the event-loop. However, the execution of the different callbacks are as distinct as the execution of the different stages of a pipeline. Precisely, the call stack is as distinct between two callbacks, as between two stages. Therefore, in the event-loop, an asynchronous function call represents the end of the call stack of the current callback, and the beginning of the call stack of the next. It represents what I call a rupture point. It is the equivalent to a data stream between two stages in the pipeline architecture.

Both the pipeline architecture and the event-loop present these ruptures points. To allow the transformation from the event-loop model to the pipeline architecture in the case of real-time web applications, I study in this thesis the possibility to transform the global memory of the event-loop into isolated memory to be able to execute the application on a pipeline architecture.

2.3.2.2 State coordination

The global memory used by the event-loop holds both the state and the data of the application. The invariance holds for the whole memory during the execution of each callback. As I explained in the previous section, this invariance is required to allow the concurrent execution of the different tasks. On the other hand, the invariance is explicit in the pipeline architecture, as all the stages have isolated memories. The coordination between these isolated process is made explicit by the developer through message passing.

I argue that the state coordination between the callbacks requireing a global memory could be replaced by the message passing coordination used manually in the pipeline architecture. I argue that not all applications need concurrent access on the state, and therefore, need a shared memory. Specifically, I argue that each state region remains roughly local to a stage during its modification. *

⚠

TODO review that, I don't know how to formulate these paragraphs. Identify the state and the data in the global memory.

2.3.2.3 Transformation

This equivalence should allow the transformation of an event loop into several parallel processes communicating by messages. In this thesis, I study the static transformation of a program, but the equivalence should also hold for a dynamic transformation. I present the analyzis tools I developed to identify the state and the data from the global memory.

With this compiler, it would be possible to express an application with a global memory, so as to follow the design principles of software development. And yet, the execution engine could adapt itself to any parallelism of the computing machine, from a single core, to a distributed cluster.

TODO too fast on the end of this section

TODO Transition to the chapter State of the Art

Chapter 3

State of the art

3.1 Javascript

Javascript was released in a hurry, without a strong and directive philosophy. During its evolution, it snowballed with different features to accommodate the community, and the usage it was made on the web. As a result Javascript contains various, and sometimes conflicting, programming paradigms. It borrow its syntax from a procedural language, like C, and the object notation from an object-oriented language, like Java, but it provides a different inheritance mechanism, based on prototypes. Most of the implementation adopt an event-based paradigm, like the DOM[1] and node.js[2]. And finally, event though it is not purely functional like Haskel, Javascript borrows some concepts from functional programming.

In this section, we focus on the last two programming paradigm, functional programming and event-based programming. Javascript exposes two features from functional programming that are particularly adapted for event-based programming. Namely, it treats functions as first-class citizen, and allows them to close on their defining context, to become closures.

3.1.1 Functions as First-Class citizens

"All problems in computer science can be solved by another level of indirection"

[1] `http://www.w3.org/DOM/`
[2] `https://nodejs.org/`

—Butler Lampson

Javascript treats function as first-class citizens. One can manipulate functions like any other type (number, string ...). She can store functions in variables or object properties, pass functions as arguments to other functions, and write functions that return functions.

The most common usage examples of these features, are the methods `Map`, `Reduce` and `filter`. In the example below, the method `map` expect a function to apply on all the element of an array to modify its content, and output a modified array. A function expecting a function as a parameter is considered to be a higher-order function. `Map`, `Reduce` and `Filter` are higher-order functions.

```
1  [4, 8, 15, 16, 23, 42].map(function firstClassFunction(element) {
2    return element + 1;
3  });
4  // -> [5, 9, 16, 16, 24, 43]
```

Higher-order functions provide a new level of indirection, allowing abstractions over functions. To understand this new level of abstraction, let's briefly summarize the different abstractions on the execution flow offered by programming paradigms. In imperative programming, the control structures allow to modify the control flow. That is, for example, to execute different instructions depending on the state of the program. Procedural programming introduces procedures, or functions. That is the possibility to group instructions together to form functions. They can be applied in different contexts, thus allowing a new abstraction over the execution flow.

So, higher-order functions add another level of abstraction. It allows to dynamically modify the control of the execution flow. The ability to manipulate functions like any other value allows to abstract over functions, and behavior.

Higher-order functions replace the needs for some Object oriented programming design patterns.[3] Though object oriented programming doesn't exclude higher-order functions.

They are particularly interesting when the behavior of the program implies to react to inputs provided during the runtime, as we will see later. Web servers, or graphical user interfaces, for examples, interact with external events of various types.

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

3.1.2 Lexical Scoping

Closures are indissociable from the concept of lexical environment. To understand the former, it is important to understand the latter first.

3.1.2.1 Lexical environment

A variable is the very first level of indirection provided by programming languages and mathematics. It is is a binding between a name and a value. Mutable like in imperative programming to represent the reality of memory cells, or immutable like in mathematics and functional programming. These bindings are created and modified during the execution. They form a context in which the execution takes place. To compartmentalize the execution, a context is also compartmentalized. A certain context can be accessed only by a precise portion of code. Most languages defines the scope of this context using code blocks as boundaries. That is known as lexical scoping, or static scoping. The variables declared inside a block represent the lexical environment of this block. These lexical environments are organized following the textual hierarchy of code blocks. The context available from a certain block of code, that is set of accessible variable, is formed as a cascade of the current lexical environment and all the parent lexical environment, up to the global lexical environment.

3.1.2.2 Javascript lexical environment

4

Javascript implement lexical scoping with function definitions as boundaries, instead of code blocks. The code below show a simple example of lexical scoping in Javascript.

```
1  var a = 4;
2  var c = 6;
3  function f() {
4    var b = 5;
5    var c = 0;
6    // a and b are accessible here.
7    return a + b + c;
8  }
9
10 f(); // -> 9
11
12 // b is not accessible here :
```

[4] http://www.ecma-international.org/ecma-262/5.1/#sec-10.2

```
13  a + b + c; // -> ReferenceError: b is not defined
```

Lexical scoping, or statical scoping, implies that the lexical environment are known statically, at compile time for example. But Javascript is a dynamic language, it doesn't truly provide lexical scoping. In Javascript, the lexical environments can be dynamically modified using two statements : `with` and `eval`. We explain in details the Javascript lexical scope in section **??**

3.1.3 Closure

"An object is data with functions. A closure is a function with data."

—John D. Cook

A closure is the association of a first-class function with its context. When a function is passed as an argument to an higher-order function, she closes over its context to become a closure. When a closure is called, it still has access to the context in which it was defined. The code below show a simple example of a closure in Javascript. The function `g` is defined inside the scope of `f`, so it has access to the variable `b`. When `f` return `g` to be assigned in `h`, it becomes a closure. The variable `h` holds a closure referencing the function `g`, as well as its context, containing the variable `b`. The closure `h` has access to the variable `b` even outside the scope of the function `f`.

```
1   function f() {
2     var b = 4;
3     return function g(a) {
4       return a + b;
5     }
6   }
7
8   var h = f();
9   // b is not accessible here :
10  b; // -> ReferenceError: b is not defined
11
12  // h is the function g with a closure over b :
13  h(5) // -> 9
```

3.1.4 Current and Future trends

ES6 / 7 TODO

Code reuse. Why it never worked ?

em-scripten

https://github.com/kripken/emscripten Javascript is a target language for LLVM, therefor everything can compile to Javascript : JS is the assembler of the web.

Isomorphic Javascript

Server-side Javascript

https://www.meteor.com/ https://facebook.github.io/flux/ Javascript can be executed both on the client and the server. That allow use-cases never possible before (server pre-rendering, same team ...)

Reactive

http://facebook.github.io/react/ Javascript is used to model the flow of propagation of state in a web application

3.2 Concurrency

Javascript, like most programming languages, is synchronous and non-concurrent. The specification doesn't provide any mechanism to write asynchronous nor concurrent execution of Javascript. There is no reference to `setTimeout` nor `setInterval`, two well-known instructions to asynchronously post-pone execution. Indeed, like for many languages, concurrency is supported and provided by the host environment. For example, the DOM for Javascript, the JVM for Java, or the operating system for C/C++. These three examples provide different models for concurrency. We explore in this section, the two different models and their evolutions.

3.2.1 Two known concurrency model

We distinguish two types of concurrent models, tailored for two types of applications, CPU bound applications, and I/O bound applications. An example of CPU bound application is a scientific application, like a physics simulation. Such applications rely heavily on the CPU to do demanding calculations. CPU bound applications need concurrency to increase the computing power and deliver a result faster. On the other hand, an example of I/O bound application is graphical user interface. It needs to display and react to multiple concurrent streams of interaction. Another example is a web server, it needs to react to multiple concurrent requests. I/O bound applications need concurrency to keep track of concurrent control flows in multiple contexts. The difference between these two needs is thin enough for

one to be mistaken with the other. The two well-known models for concurrency are threads and events. It is now broadly admitted that threads are the better solution for CPU bound applications, while events are the better solution for I/O bound applications. However, this distinction is very blurry, because of a lack of precise definitions for both models. In the next section I explain what characterizes these two models and what are the differences between the two.

when there is the need for concurrency in a system (so the components are somewhat dependent on each others : causal relation, or state sharing), there is a need for coordination. If there is no need for coordination, then there is two independent systems. So concurrency is first a topology and coordination (communication?) problem. Synchronization is a type of coordination, asynchronous messaging is another type of coordination.

3.2.1.1 Thread

A thread is the lightest representation of a sequence of instructions executed on a computing core. Multiple threads can be executed concurrently. Threads are not to be mistaken with processes. A process always contain at least one thread, sometimes more. Multiple threads share the same memory, while multiple processes don't. The thread is a very broad concept, it generalize kernel threads, green threads or user threads and fibers[**Adya2002**].

A kernel thread is a set of registers on a computing core representing the state of a computation[5]. One of these registers being the instruction pointers, which is manipulated by the control flow, a thread indeed represent the execution of a sequence of instructions on this computing core. It is the smallest execution unit manipulable by the kernel scheduler. C/C++ uses kernel threads provided by the operating system.

A green thread, or user thread, is the implementation of a thread in user level, instead of kernel level. Java uses green threads provided by the JVM.

Threads, and particularly kernel threads are often preemptively scheduled. A fiber, on the other hand, is a thread non preemptively scheduled but cooperatively scheduled. I address in the next section exactly what is a preemptive or cooperative scheduler.

[5] `http://stackoverflow.com/a/5201879/933670`

3.2.1.2 Event

An event is a significant change in the [application] state[**Chandy2006**]. In an event-based system, the concurrent executions are independent, they don't share any memory. To coordinate the execution, they exchange messages representing the occurrence of events impacting the execution at a larger scale than the local concurrent execution.

The actors model is an example of an event-based concurrent model for Artificial Intelligence purposes [3].

3.2.1.3 Orthogonal concepts

Threads doesn't indicate anything about synchronization, but focus solely on execution. Events doesn't indicate anything about execution, but focus solely on synchronization. They are orthogonal concepts. It is completely pointless to compare them.

There is two differences between threads and events. The memory sharing, and the programming style. Threads share memory, while events don't, so threads needs synchronization to coordinate. Threads use synchronous programming, while events use asynchronous programming.

Events with shared memory is an event-loop : there is no parallelism.

Lauer and Needham [**Lauer1979**] presented an equivalence between Procedure-oriented Systems and Message-oriented Systems.

Adya *et. al.* analyzed this debate and presented fives categories through which to present the problem [**Adya2002**]. Their advantages and drawbacks were mistaken with those of thread and events. Adya *et. al.* explain in details two of these categories that are most representative, Task management and Stack management. These two categories were often associated with thread-based systems and event-based systems. We paraphrase these explanations.

Multi-process programming use Inter Process Communications (IPC). IPC regroups both shared memory(?) coordination, and message passing(?) coordination. Semaphore, shared memory, and files are shared-memory coordination mechanisms, while pipe, streams and message queues, are message passing coordination mechanisms.

Time-slicing is used when there is more threads than cores Multi-threading is used when there is multiple cores. These cases are not exclusive. In each cases, there is different problematics.

3.2.2 Differentiating characteristics

Automatic stack management is as bad as preemptive scheduling : you don't know when the preemption happens. It is in reality impracticable to use cooperative task management with automatic stack management. Ayda said it : with closure, the problem of stack management disappear. It remains only the difference between sync / async programming : the control flow is broken in multiple handlers (async) or appear to be continuous (sync). It is linked with the task management. Cooperative task management provides a known invariant. While preemptive task management hides the invariant. Actually, it is neither task management, nor stack management, it is knowing yielding points, to know the invariant.

3.2.2.1 Scheduling

3.2.2.2 Coordination strategy

3.2.3 Turn-based programming

When an event-loop is used as the concurrency model for a partially functional language like Javascript, it creates what Douglas Crockford called turn-based programming[6].

3.2.3.1 Event-loop

3.2.3.2 Promises

3.2.3.3 Generators

Generators might be used to implement synchronous programming on top of an event-loop (asynchronous programming), a bit like fibers Generators use the yield keyword, which is good stuff. The long term goal would be to have asynchronous yield as rupture points.

However, one very important thing, is that yield block the execution. And if it is a good thing for programmers to know when the execution is leaving the atomicity (yield vs fibers), I want to explore how it might reduce the concurrency expressiveness. yield impose a linear programming style, while promises keep the tree programming style.

[6] `https://youtu.be/dkZFtimgAcM?t=1852`

3.2.4 Message-passing / pipeline parallelism -> DataFlow programming ?

"One early vision was that the right computer language would make parallel programming straightforward. There have been hundreds—if not thousands—of attempts at developing such languages ... Some made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming. Nor has any become as popular as the languages invented primarily for sequential programming."

—David Patterson

3.2.4.1 TODO

3.3 Scalability

3.3.1 Theories

3.3.1.1 Linear Scalability

3.3.1.2 Limited Scalability

3.3.1.3 Negative Scalability

Conclusion : scalability = concurrency + not sharing the resources that grows with the scale

3.3.2 Scalability outside computer science (only if I have time)

If I have time, I would like to try to explain why scalability is at the core of material engagement and information theory, and is at the core of our universe : the propagation of Gravity wave is an example : it is impossible to scale

3.4 Framworks for web application distribution

3.4.1 Micro-batch processing

3.4.2 Stream Processing

3.5 Flow programming

3.5.1 Functional reactive programming

3.5.2 Flow-Based programming

3.6 Parallelizing compilers

OpenMP and so on

3.7 Synthesis

There is no compiler focusing on event-loop based applications

Chapter 4

Fluxion

4.1 Fluxionnal Compiler

Some parts of this are already written in the first paper. It needs a lot additional explanations and rewritting

4.1.1 Identification

4.1.1.1 Continuation and listeners

4.1.1.2 Dues

4.1.2 Isolation

4.1.2.1 Scope identification

Scope leaking

4.1.2.2 Execution and variable propagation

4.1.3 distribution

4.2 Fluxionnal execution model

Everything here is already written in the first paper : flx-paper. It only needs to be rewritten

4.2.1 Fluxion encapsulation

4.2.1.1 Execution

4.2.1.2 Name

4.2.1.3 Memory

4.2.2 Messaging system

Chapter 5

Evaluation

5.1 Due compiler

5.2 Fluxionnal compiler

5.3 Fluxionnal execution model

Chapter 6

Conclusion

Appendix A

Language popularity

A.1 PopularitY of Programming Languages (PYPL)

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

PYPL for May 2015

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

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

A.2 TIOBE

[3]

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

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

TIOBE for April 2015

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

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

A.3 Programming Language Popularity Chart

[4]

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

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

indeed.com

A.4 Black Duck Knowledge

[5]

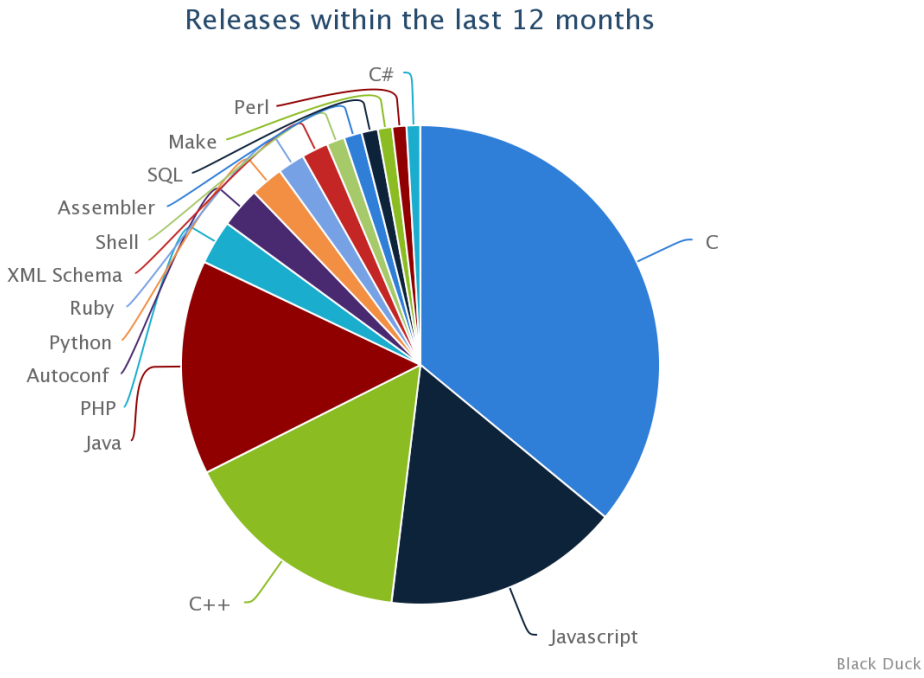
The black-duck, which analyze the usage of language on many forges, and collaborative hosts, rank Javascript number 2, after C, and with about the same usage as C++.

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

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

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

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



A.5 Github

http://githut.info/

A.6 HackerNews Poll

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

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

Bibliography

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

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

[3] C Hewitt, P Bishop, and R Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint ...* (1973).