Les performances Web avec LLVM et WebAssembly

28 octobre 2022

webperformance

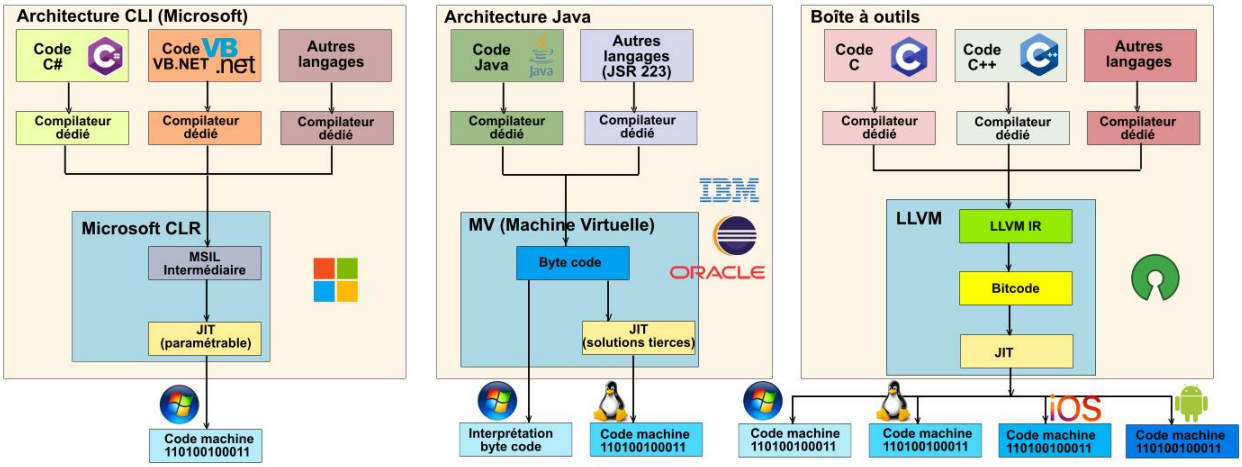
claude@lemarson.com
https://www.lemarson.com

Sommaire

Tout pour le client Web

L'architecte du TI est face à un problème majeur, celui d'améliorer la performance et la portabilité sans efforts des applications portées par les navigateurs serveurs et clients.

Le contexte : Open Source vs propriétaire



Tendance : webisation des applications

(le web partout)

- Années 90 : l'architecture Web ne se retrouvait que dans les sites, avec le trio HTML, les feuilles de style et JavaScript.
- Les clients lourds de la même époque ont fait long feu (mode client/serveur) :
  - Difficiles à maintenir.
  - Difficultés à installer : il manquait toujours un composant.
  - Dépendance vis-à-vis de Microsoft.
  - Performances limitées.
- Les clients "riches" HTML/JavScript ont commencé à s'imposer qui se sont substitués aux clients lourds :
  - Les machines et les réseaux se sont fortement améliorés.
  - Des API nouvelles fondées sur des frameworks "intelligents" ont pu étendre les possibilités de JavaScript.
  - Des "transpilers" (compilateurs de source à source) sont apparus, qui ont apporté du sucre syntaxique là où JavaScript était le plus insuffisant.
  - HTML 5 a été annoncé en 2014 : bond très important du langage déclaratif.
  - JavaScript a été normalisé avec ECMAScript, ce qui a rassuré les utilisateurs potentiels.
  - Le paysage JavaScript lui-même s'est simplifié avec le retrait effectif de VBScript chez Microsoft.
- La plupart des applications, Web, mais aussi facturation, gestion des stocks, gestion clients, etc, ont remplacé leur client lourd par un client JavaScript.
- Le principe s'est répandu sur la partie serveur, avec les initiatives de type Node.

LLVM et WASM dans un processus...

...rendu nécessaire

Exploiter un véritable langage dans le navigateur, titre que ne revendique pas JavaScript. Capacités fonctionnelles, par exemple.

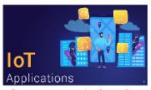
Exécuter du code plus rapidement dans le navigateur, que JavaScript.

Produire du code cross-platform, sans adaptation, ni recompilation.

Faciliter la maintenance d'applications dont le client fonctionne dans un navigateur.

Supporter une charge de travail imprévisible sans écrouler le système. JavaScript n'a pas cette vocation.

Augmenter les fonctionnalités natives de JavaScript, pour répondre aux besoins nouveaux.



Prendre en compte les domaines mobiles et IoT. A priori, LLVM et WASM ne sont pas adaptés aux capteurs. Mais cela peut changer.

Prendre en compte les applications "monpages", que l'on retrouve de plus en plus.

Partage des applications déportées dans le Cloud. Sans adaptation ni réécriture.

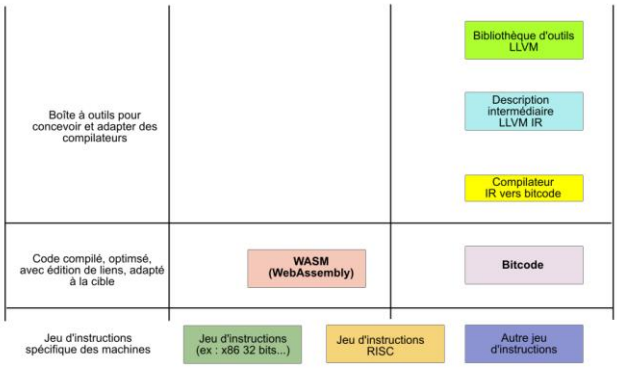
Récupérer les "vieilles" applications, que l'on veut améliorer, sans les réécrire.

Assurer des fonctions d'optimisation, incluses dans le processus d'adaptation et de portage.

LLVM et WASM (WebAssembly)

La grande différence : WASM est un format de langage intermédiaire, alors que LLVM est une boîte à outils pour fabriquer des compilateurs.

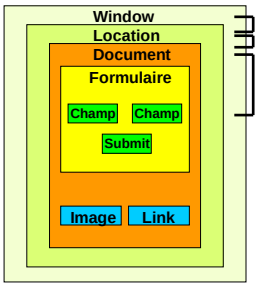
- LLVM est un projet "privé" démarré en 2000 à l'université de l'Illinois, pour développer des techniques de compilation dynamique. En 2005, Apple embauche l'un des acteurs de LLVM, Chris Lattner, et forme une équipe pour intégrer les outils dans Xcode.
- WASM est un format ouvert de langage intermédiaire, proposé par le W3C depuis 2017 qu'il faut adapter aux plates-formes. C'est une cible de compilation. Plusieurs acteurs ont contribué à son élaboration : Mozilla, Microsoft, Google, Apple, Intel, RedHat.
- On peut compiler l'IR de LLVM en WebAssembly.



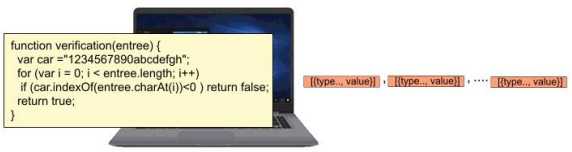
Objectifs : performances, portabilité, indépendance.
Un problème d'architecture majeur aujourd'hui.
Sauf à travailler chez un éditeur ou un prestataire spécialisé, l'utilisateur n'aura pas à s'investir dans les technologies WASM et LLVM, mais il devra connaître les points essentiels pour comprendre les implications de ces technologies sur le reste du TI. Attention à la prise d'otages...

Tout part de JavaScript

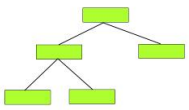
- Personne n'aurait imaginé que JavaScript aurait le succès qui a été le sien.
- A l'origine, son créateur Brendan Eich, voulait seulement améliorer la présentation.
- Fondé sur la hiérarchie DOM (Document Object Model) : le document est un objet constitué d'autres objets tels que des formulaires, des images, des tables, des boutons, qui possèdent des propriétés et réagissent à des événements par des méthodes.
- Ses principaux usages :
  - Validation de formulaires.
  - Embellissement de pages web et insertion d'effets spéciaux : changement de couleur d'une image (roll over)…
  - Quelques fonctions mathématiques de base.
  - Traitement local de textes, XML ou autres formats, issus d'un traitement serveur : technologie Ajax.
  - Connaître l'identité du navigateur local grâce aux propriétés de l'objet Navigator.
- Ce n'est pas un langage simple, il est relativement dangereux et supporte une certaine « mauvaise réputation ».
- La première avancée a été le moteur V8 de JavaScript Google Chrome. L'idée étant de compiler le code JavaScript en langage machine, via une procédure JIT et optimisation (2008).
- Une autre date importante : 2012. C'est Alon Zakai de Mozilla qui lance une idée originale et "saugrenue" de faire tourner un code C dans le navigateur. Il crée pour cela Emscripten.



La "mécanique" AST
(rappel)



Phase 1 : analyse lexicale, génération d'une liste tokens



Phase 2 : analyse syntaxique (parsing), génération d'une arborescence

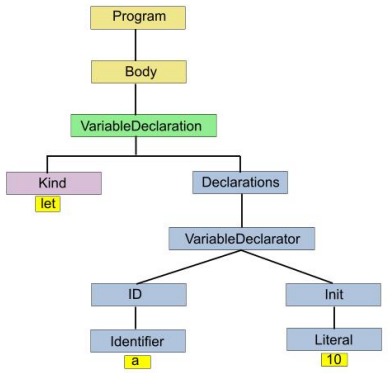
Phase 3 : génération du code exécutable ou source (transpilage)

La "mécanique" AST

- Le programme JavaScript est un flux, qui est traduit en "tokens" : mots réservés, identificateurs, littéraux, ponctuation, tout sauf les espaces.
- Ce sont ces tokens qui sont envoyés au parser pour analyse et fabrication d'un arbre syntaxique (AST), dont les nœuds sont les tokens.
- L'interpréteur navigue dans l'AST et génère un byte code, code intermédiaire, qui remplace l'AST.
- Le profiler surveille le code et l'optimise.
- Le compilateur génère du code de plus bas niveau, compréhensible par la machine.
- Optimisation.

`let a = 10`

Le parser identifie quatre éléments : **let** (keyword/Kind), **a** (Identifier), **=** (Assignment) et **10** (Literal)




```
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
        {
          "type": "VariableDeclarator",
          "id": {
            "type": "Identifier",
            "name": "a"
          },
          "init": {
            "type": "Literal",
            "value": 10
          }
        }
      ],
      "kind": "let"
    }
  ]
}
```


Dans cet exemple (très simple), chaque bloc est un "node", qui commence avec la déclaration du Program, celui-ci étant constitué d'un Body dans lequel sont déclarés les nodes. La déclaration de variable VariableDeclaration concerne a, dont la description est divisée en deux : Kind (type de a) et Declarations, avec deux nodes ID et Init. Le fichier JSON est une instanciation de l'arborescence, exploitable par les codes en aval : compilateur, interpréteur, transpiler et applications.

Le moteur v8 JavaScript de Google

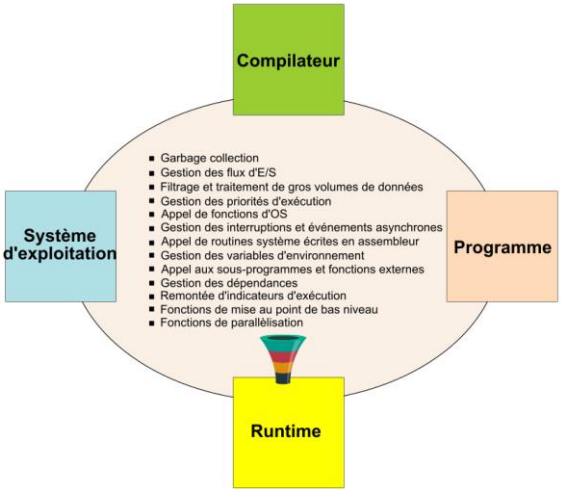
- V8 fonctionne en mode JIT (Just In Time) et compile le code .js, en l'optimisant grâce aux techniques de « code inlining » ou de « copy elision » (technique d'optimisation qui évite de copier les objets inutiles).
- Développé pour Chrome, pour améliorer les performances JavaScript (Lars Bak en 2008)
- Ce que fait (bien) V8 :
  - Compile le code JavaScript.
  - Gère les call stacks (pour garder la position des appels de fonctions en cas d'appels multiples).
  - Gère la mémoire assignée aux programmes ("heap memory").
  - Garbage collection.
  - Compatible avec tous les types de données.
- Elément essentiel de V8 : "ingnition interpreter", qui compile le code JS en code machine non optimisé.
- A l'exécution, le code machine est analysé et recompilé pour obtenir les meilleures performances, via deux composants clés : TurboFan et Carnkshaft.
- En 2018 : V8 Lite, dont les éléments de simplification sont portés dans V8.
- En moyenne : 18 % d'amélioration pour les sites Web classiques.

Inlining : remplacement par le compilateur du code d'un appel de fonction par le code de la fonction appelée. Ce qui fait disparaître les appels de fonctions. Le compilateur doit évaluer si cela présente un intérêt : la taille du code peut augmenter sensiblement, le code peut être plus lent et consommer plus de ressources. Tout dépendra des fonctions. L'inlining n'est pas seulement une technique d'optimisation, c'est un encouragement à écrire du code optimisé, bien découpé.

Le concept de runtime : positionnement

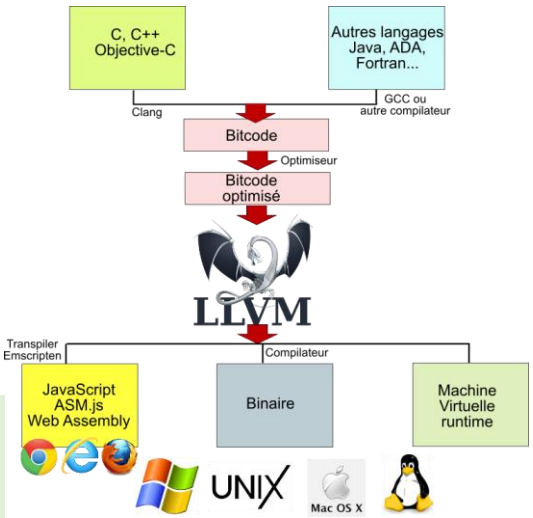
L'importance accrue de l'optimisation du code et des "run time"

- Sujet dont on ne parle pas de manière générique, mais toujours en relation avec un produit.
- Les performances des systèmes modernes s'expliquent largement par les runtimes.
- Le runtime est une plate-forme matérielle et/ou logicielle dédiée à un langage, à un SGBD, à une API, etc, qui leur apporte des services.
- Peut être une MV, un container…
- Il est sollicité uniquement pendant la phase d'exécution.
- Pour un langage, le runtime est constitué de l'ensemble des "comportements" qui ne sont pas pris en charge par le programme lui-même.
- Il y a aussi les engins liés à une API (architecture), aux bases de données (SGBD), aux compilateurs SQL (PL/SQL), etc.
- Deux grandes familles :
  - Java, .NET et Python, pour leur machine virtuelle.
  - Les langages script : JavaScript, PHP…



La boîte à outils LLVM et son architecture

- LLVM (Low Level Virtual Machine) est un ensemble d'outils écrits en C++ qui permettent d'écrire du code portable et proche du langage machine à la compilation.
- LLVM a été imaginé pour produire des outils de compilation originaux et performants, susceptibles de générer du code intermédiaire, dit « bitcode », à partir de n'importe quel langage. Et donc d'assurerleur portabilité, à condition de prévoir un environnement d'exécution du dit bitcode sur les machines-cibles.
- LLVM est au départ un projet de génération d'un code de bas niveau, proche des machines, comme peut l'être un byte-code Java.
- Deux niveaux : IR (intermédiaire) et bitcode, l'IR jouant un rôle de backbone fédérateur pour accueillir des sources distinctes de codes, issues de plates-formes différentes.
- La solution imaginée avec LLVM avait pour objectif d'être plus performante que le byte code Java et mieux optimisée.
- Mais sur le principe l'idée était la même.



LLVM est constitué d'un ensemble d'outils qui facilitent l'écriture de ces compilateurs, un traducteur de code source en bitcode, un compilateur de bitcode en langage machine, l'équivalent des JIT (Just-In Time) de .NET et Java, mais aussi un éditeur de liens (linker), un désassembleur, un outil de mise au point LLDB, un optimiseur de code, des machines virtuelles JIT, un interpréteur.
Toute une galaxie d'outils fondés sur ce code intermédiaire de bas niveau, qui continue d'évoluer avec une communauté qui s'enrichit et constitue désormais un véritable écosystème, avec de nombreux projets de recherche, dont une centaine estampillés « Open Source », mais aussi des réalisations concrètes chez des éditeurs…plus propriétaires.

LLVM la "boîte à outils" universelle

- LLVM comporte des primitives indépendantes de l'architecture des machines, qui cachent la diversité et la complexité des plates-formes : des entiers de longueur quelconque, des moyens pour générer un code compatible avec n'importe quel jeu d'instructions.
- L'usage de LLVM par les langages :
  - La plupart utilisent LLVM en tant que compilateur amont (AOT : "Ahead-Of-Time"), ex de Clang.
  - D'autres pour le "just-in-time", compilation à l'exécution : Julia qui a besoin d'un code rapide pour interagir avec le mode REPL : "Read-Eval-Print-Loop" (prompt interactif), pour la compilation de requêtes SQL avec PostgreSQL (cinq fois plus rapides).
  - LLVM peut être utilisé pour l'optimisation du code : fonctions d'inlining, suppression de code mort, traitement des boucles ("unrolling loops")
  - Transpiler JavaScript avec le projet Emscripten (conversion de bitcode en JavaScript).
  - Pour étendre le périmètre d'un langage, sans le remettre en cause : ex du Nvidia CUDA Compiler, qui ajoute le support CUDA à d'autres langages.
- Dernière version : 15.0.3 du 18 octobre 2022.

```
int arith(int x, int y, int z){
      return (x*y + z);
}
```

Code C ou Java

```
define i32@arith(i32 %x, i32 %y, i32 %z){
      entry
      %tmp = mul i32 %x, %y
      %tmp2 = add i32 %tmp, %z
      ret i32 %tmp2
}
```

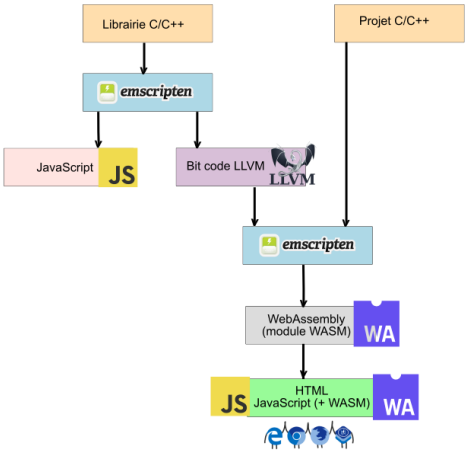
LLVM IR

```
public class demo.Demo {
      public static int arith(int, int, int);
            Code:
                  0: iload_0
                  1: iload_1
                  2: imul
                  3: iload_2
                  4: iadd
                  5: ireturn
}
```

Bytecode Java

WASM : architecture et format cible

- Le WebAssembly (WASM), est un formatage de bas niveau, dont l'objet est d'exécuter, à grande vitesse, des applications dans un navigateur Web.
- Résultat de la compilation d'un code écrit avec un langage déjà proche de la machine, comme C, C++ ou Rust.
- La raison est la gestion mémoire : en JavaScript, Java et d'autres langages de « haut niveau », la gestion mémoire s'effectue via un « garbage collector » ou équivalent, qui prend à sa charge le « nettoyage » de l'espace mémoire, alors qu'avec C, C++ ou Rust, il faut tout gérer manuellement. WebAssembly fonctionne de cette façon, ce qui le rend plus facile à produire en compilation avec C, C++ ou Rust, qu'avec un autre langage.
- Très proche de JavaScript, on peut appeler des modules WebAssembly à partir d'un code JavaScript, ils sont complémentaires.
- WASM est un standard W3C. Il date de 2015.



WASM

- S'appuie sur deux éléments : une machine virtuelle spécifique, capable d'exécuter du WebAssembly ET du JavaScript et une API, des fonctions que le navigateur va appeler pour faire fonctionner l'application.
- L'un des avantages du WebAssembly est qu'on peut le générer à partir de plusieurs langages de haut niveau : les langages proches, mais aussi d'autres qui font l'objet de nombreux projets.
- On peut par exemple écrire du code C, C++ ou Rust, que l'on va compiler avec Emscripten ou d'autres plates-formes LLVM.
- Le transpiler TypeScript, peut aussi produire du WebAssembly.
- En Java, il faut s'intéresser à Bytecoder et surtout à TeaVM, qui traduit du bytecode Java en WebAssembly, de même qu'à partir du code intermédiaire Kotlin.
- Supporté par la quasi-totalité des navigateurs,(95 % en septembre 2022), y compris pour iOS et Android.
- Compatible avec 40 langages, qui génèrent du WASM.
- Pour les vieux navigateurs, peut être compilé en asm.js, via un "polyfill" JavaScript : simulation des fonctions non disponibles nativement.
- Il existe des implémentations de WASM hors navigateur : Wasmer, Wasmtime, wasm3, WAVM et de nombreuses autres.

Le format WASM

- Pas destiné à être lu par les développeurs, comme le byte code Java, mais…
- Jeu d'instructions : autour de 200 en standard, mais il y a des spécialisations, comme le mode SIMD (traitement parallèle) : 236 codes préfixés.

C source code and corresponding WebAssembly

C source code	WebAssembly `.wat` text format	WebAssembly `.wasm` binary format

`int factorial(int n) {`
`  if (n == 0)`
`  return 1;`
`  else`
`  return n * factorial(n-1);`
`}`
	`(func (param i64) (result i64)`
	`  local.get 0`
	`  i64.eqz`
	`  if (result i64)`
	`  i64.const 1`
	`  else`
	`  local.get 0`
	`  local.get 0`
	`  i64.const 1`
	`  i64.sub`
	`  call 0`
	`  i64.mul`
	`  end)`
		`00 61 73 6D 01 00 00 00`
		`01 00 01 60 01 73 01 73 06`
		`03 00 01 00 02`
		`0A 00 01`
		`00 00`
		`20 00`
		`50`
		`04 7E`
		`42 01`
		`05`
		`20 00`
		`20 00`
		`42 01`
		`7D`
		`10 00`
		`7E`
		`0B`
		`0B 15 17`

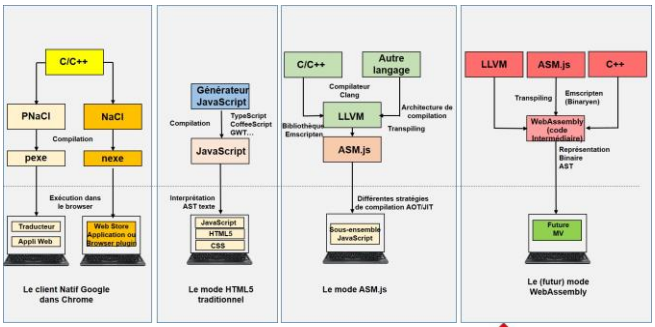
Wikipedia

Le format texte de WASM est destiné aux éditeurs de textes, aux outils de développement dédiés au Web, etc. Il ne concerne pas l'utilisateur final, même s'il est lisible.
Dans les deux cas, binaire et texte, l'unité de code WASM est un module, qui exploite une vieille technologie, mise en avant par le langage LISP, les "S-expression" pour représenter des arborescences par un formalisme "à plat".

Des solutions qui aboutissent à WASM

(ou qui se servent de WASM)

Les performances du client Web



Inconvénients de LLVM et WASM

- Le sujet n'est pas neutre et les choix d'architectures doivent être validés par le TI client : les conséquences financières et techniques peuvent être très lourdes.
- Fait partie des domaines à connaître, même si on ne "fait pas"…
- Il ne faut pas le laisser au choix des prestataires, qui n'ont pas nécessairement les mêmes contraintes et besoins.
- Sauf à travailler chez un éditeur ou un prestataire spécialisé, l'utilisateur n'aura pas à s'investir dans ces technologies, mais il doit connaître les points essentiels pour comprendre leurs implications sur le reste du TI.

WASM

- En général, WASM n'interagit pas avec le DOM JavaScript et les interactions doivent être implémentées en JavaScript classique.
- Il n'y a pas de ramasse miettes ("garbage collector")… pour l'instant.
- Des failles importantes en termes de sécurité : phishing, difficultés pour analyser le code compilé, cryptominage (affaire Coinhive, aujourd'hui disparu)

LLVM

- Pas de ramasse-miettes, qui doit être implémenté par le runtime d'accueil.
- Le code LLVM IR n'est pas portable.
- Difficultés d'appréhension du code LLVM IR.
- Mise au point difficile.
- Le processus d'adaptation après compilation du code généré depuis LLVM IR, est complexe et demande beaucoup de temps.
- L'essentiel des développements est financé par Apple, Qualcomm et Google…