# JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications

Julien Pagès

ICPS/CAMUS

5 Avril 2017

# Le papier

JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications.
Yeoul Na, Seon Wook Kim, Youngsun Han, TACO Janvier 2016.

Mot-clefs :

- JavaScript
- JIT
- javascriptcore
- loop parallelization
- javascript engines
- HTML5

# Contexte

## Historique

- Langage de script (à la base) côté client
- De plus en plus de code JavaScript dans les sites

## Exploiter le parallélisme

- JS est historiquement optimisé pour du mono-thread
- Langage dynamique avec beaucoup de problèmes pour être parallélisé

# Langage JavaScript

Créé en 1995 par Brendan Eich.

## Caractéristiques

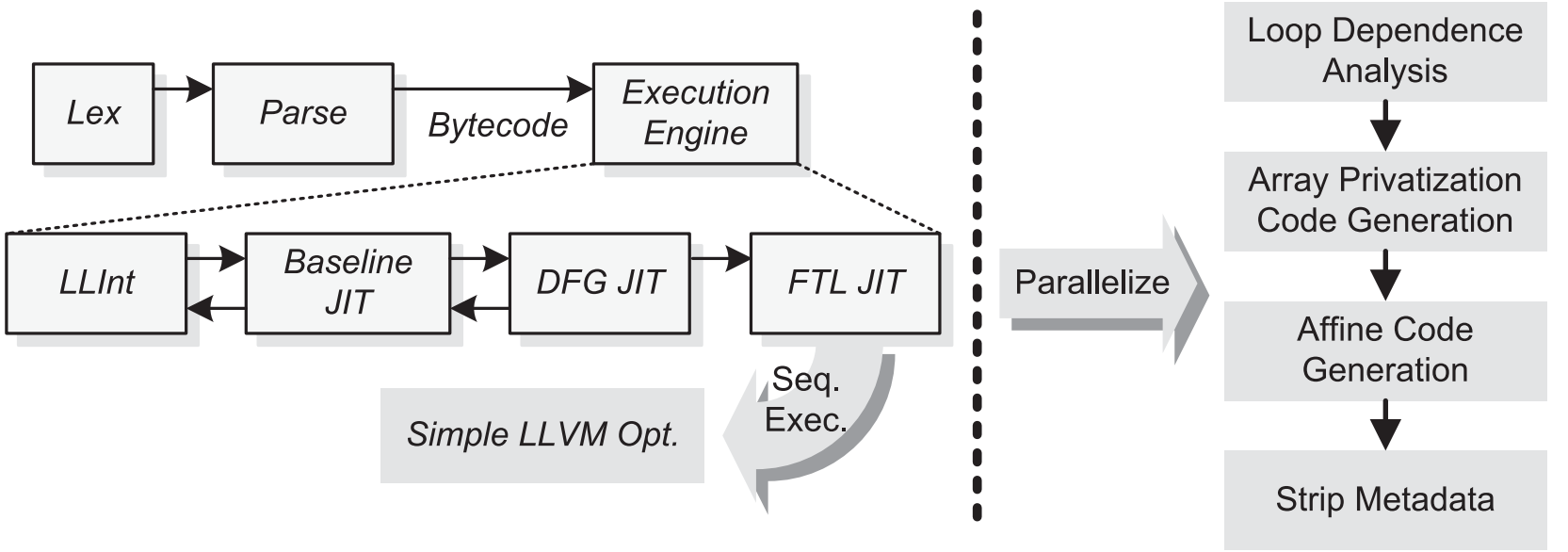
- Langage pour le web côté client
- Typage dynamique
- Orienté script
- Fonctionnel et impératif
- Orienté objet avec des prototypes

Norme ECMAScript-262 pour la spécification du langage.

# Outils utilisés

1. VM Javascript core dans Webkit
   - VM JavaScript d'Apple intégrée dans Safari
   - Interprétation + compilation à la volée
   - Plusieurs niveaux de compilation du code
2. Paralléliseur AESOP
3. Modification de la VM et du paralléliseur

# Fonctionnement de Javascript core




(a) JavaScriptCore

(b) Parallel tier extension

## OSR-exit et OSR-entry

Une optimisation : niveau suivant.

Une dé-optimisation : niveau précédent.

# Parallélisation de boucles

Modèle de coût : les boucles dans FTL JIT sont candidates.

## Boucles candidates

Seules les boucles sans dépendances sont parallélisées : **DOALL**.
Seules les boucles *idempotentes* sont parallélisées : pas de **checkpoints**.

## Problème

Chaque objet JavaScript est accédé via des pointeurs : difficultés pour l'analyse de dépendances avec l'aliasing.

## Exposition des informations sur les accès tableaux

```
1  %obj = load i64* inttoptr (i64 4334912064 to i64*)
2  %payload = add i64 %obj, 8
3  %payloadptr = inttoptr i64 %payload to i64*
4  %base = load i64* %payloadptr
5  %index = load i32* %0
6  %indexz = zext i32 %index to i64
7  %index_s3 = shl i64 %indexz, 3
8  %elemaddr = add i64 %base, %index_s3
9  %elemaddrptr = inttoptr i64 %elemaddr to i64*
10 store i64 %indexz, i64* %elemaddrptr
```

(a) Original IR

```
1  %obj = load i64* inttoptr (i64 4334912064 to i64*)
2  %payload = add i64 %obj, 8
3  %payloadptr = inttoptr i64 %payload to i64*
4  %base = load i64* %payloadptr
5  %index = load i32* %0
6  %indexz = zext i32 %index to i64
7  %baseptr = inttoptr i64 %base to [10000 x i64]*
8  %scevgep = getelementptr [10000 x i64]* %baseptr, i64 %indexz
9  store i64 %indexz, i64* %scevgep
```

(b) Enhanced IR

# Rollbacks

chaque objet JavaScript a ses propres propriétés sans alias

## Spéculation sur les pointeurs : inspecteur exécuteur

- Un test entre chaque paire de pointeurs est inséré avant l'entrée dans la boucle parallèle
- Si un alias est détecté, on n'exécute pas la boucle parallèle

Pas de rollbacks dû au parallélisme mais avec les OSR-exits :

- Changement des types des variables
- Accès à un "trou" du tableau
- Overflow d'une addition d'entiers
- ...

# Rollbacks

## OSR-exit dans un thread parallèle :

- Seul le thread principal gère les OSR-exit
- Les threads parallèles communiquent l'OSR-exit au principal

# Rollbacks

## OSR-exit dans un thread parallèle :

- Seul le thread principal gère les OSR-exit
- Les threads parallèles communiquent l'OSR-exit au principal

## Idempotence

La boucle est ensuite re-exécutée en séquentiel depuis le début.

# La paralléliseur

## AESOP

Un paralléliseur statique pour de l'IR LLVM : pas de polyédrique, pas très puissant mais léger.

Intégré dans Javascript Core sous forme de **passes LLVM**

# Benchmarks

- RiverTrail : librairie et machine virtuelle d'Intel pour des tableaux parallèles
- Pixastic : librairie de traitement d'image
- Polybench manuellement réécrits en JavaScript
- Quelques applications réelles

# Résultats

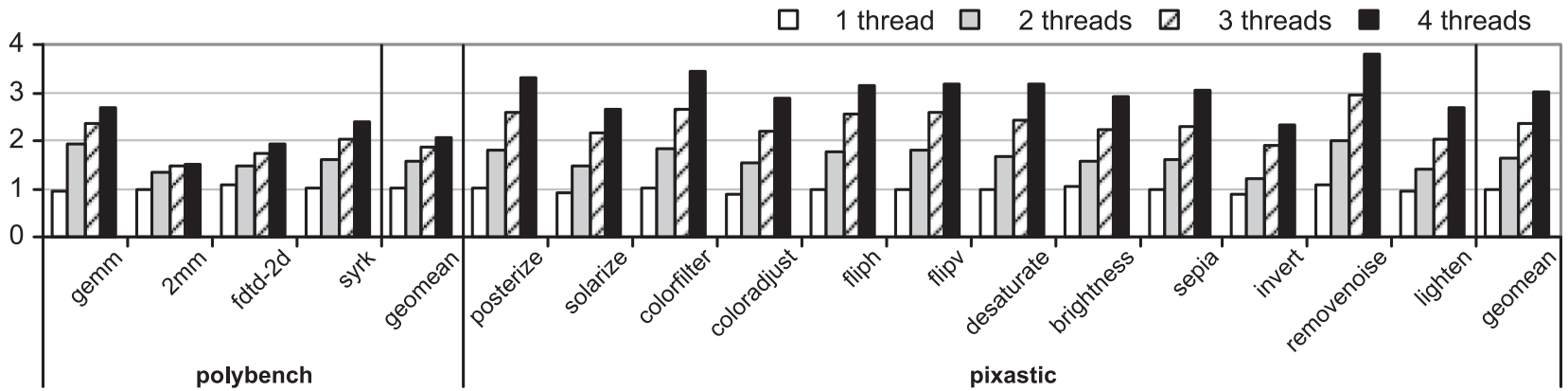



Figure: Speedup en fonction du nombre de threads

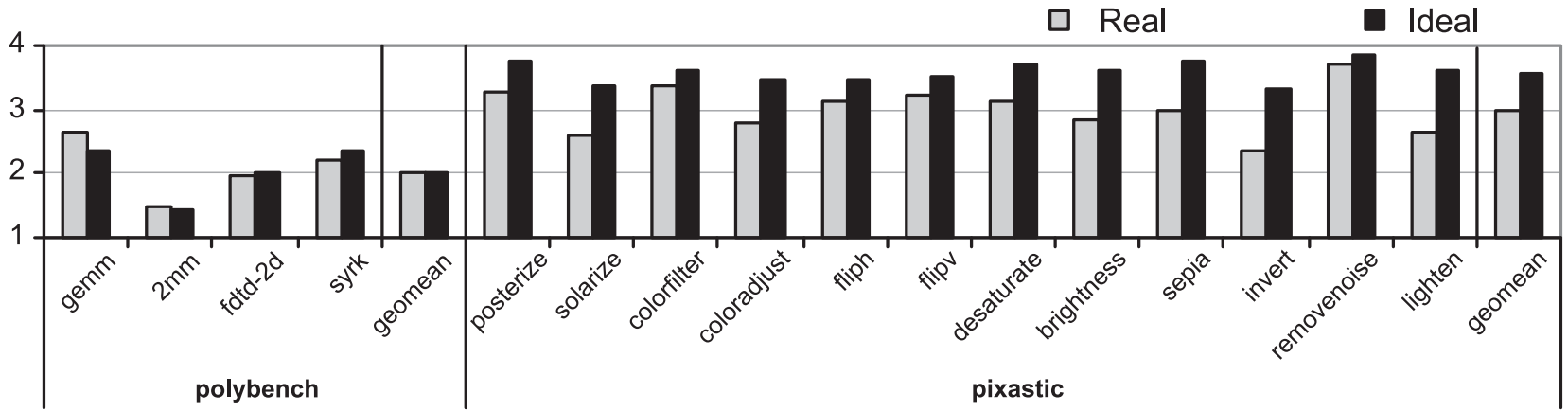



Figure: Speedup idéal contre speedup réel avec 4 threads

# Résultats

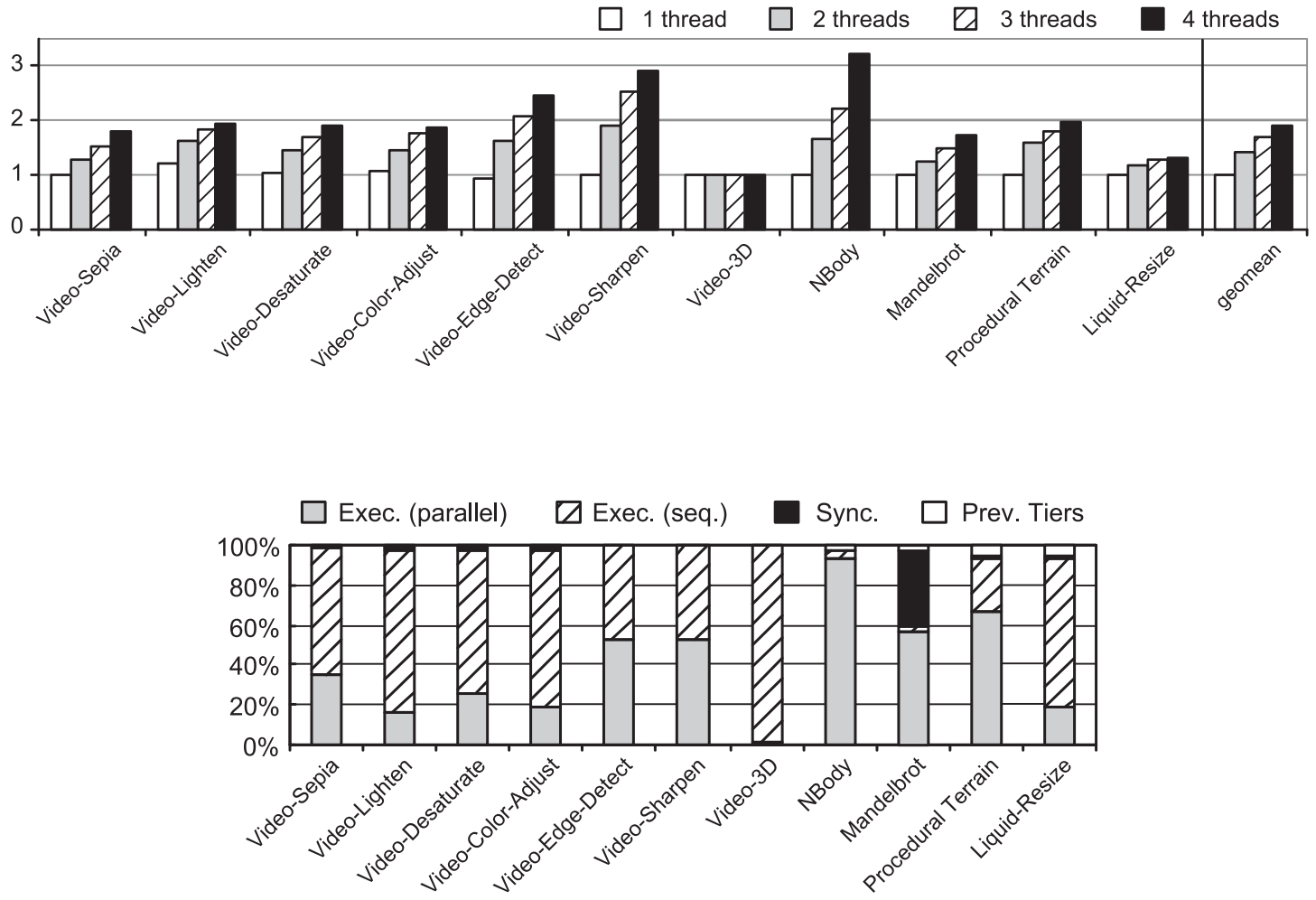



Figure: Speedups applications HTML5

# Bilan

## Résultats

- Approche innovante de parallélisation automatique en JavaScript.
- Bons speedups obtenus
- Faible surcoût des rollbacks : au plus 5%.

# Problèmes de l'approche précédente

- Peu de boucles sont candidates à la parallélisation : DOALL + idempotence
- Mécanismes de rollbacks réexécutant les boucles
- Support des boucles sur un tableau à une dimension seulement
- Inspecteur-exécuteur coûteux

## AESOP

- Pas de véritable publication autre qu'un *white paper*.
- On se sait pas vraiment ce que fait ce paralléliseur

# Questions ouvertes

1. Est-ce qu'un paralléliseur polyédrique est utilisable dans une VM au runtime ?
2. Quels sont les gains ?
3. Parallélisation des boucles à deux dimensions ?
4. Parallélisation automatique poussée et langage dynamique

# Objectifs

Utilisation de Webkit et JavascriptCore pour son backend LLVM compatible avec des outils polyédriques.

## Plan

1. Modification de l'IR pour exposer les accès tableaux
2. Dump de l'IR et tests en mode "hors-ligne" avec Polly
3. Intégrer Polly/Apollo dans JSC
4. Expérimentations

# État du projet

- Modification du backend pour utiliser `getelementptr`
- Sortie des bases des tableaux du coeur de la boucle (base invariante)
- Suppression de quelques OSR-Exit dans les boucles pour simplifier le CFG
- Tests en mode hors-ligne avec polly

Merci pour votre attention