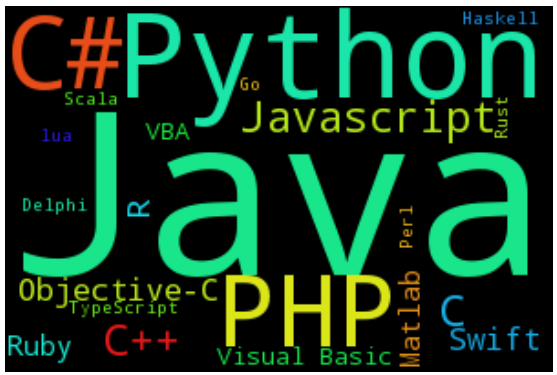
Optimisation polyédrique à l'exécution dans le contexte de langages dynamiques

Julien Pagès, **Manuel Selva**, Philippe Clauss

INRIA CAMUS, ICube, CNRS, Université de Strasbourg

21 juin 2017

Motivations

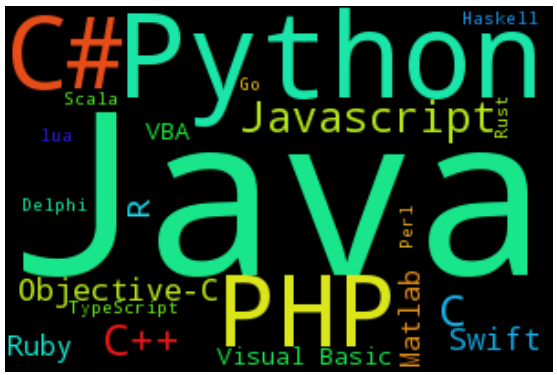


5ème langage - http://pypl.github.io

Motivations

Historique JavaScript

- 1995 - Création chez Netscape
- 1997 .. 2016 - Norme ECMAScript
- Aujourd'hui
  - Côté client et serveur
  - Pour des applications complexes



5ème langage - http://pypl.github.io

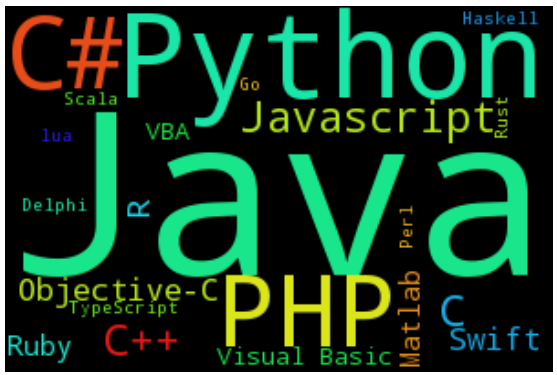
Motivations

Historique JavaScript

- 1995 - Création chez Netscape
- 1997 .. 2016 - Norme ECMAScript
- Aujourd'hui
  - Côté client et serveur
  - Pour des applications complexes

Caractéristiques

- Orienté objet
- Non typé
- Dynamique

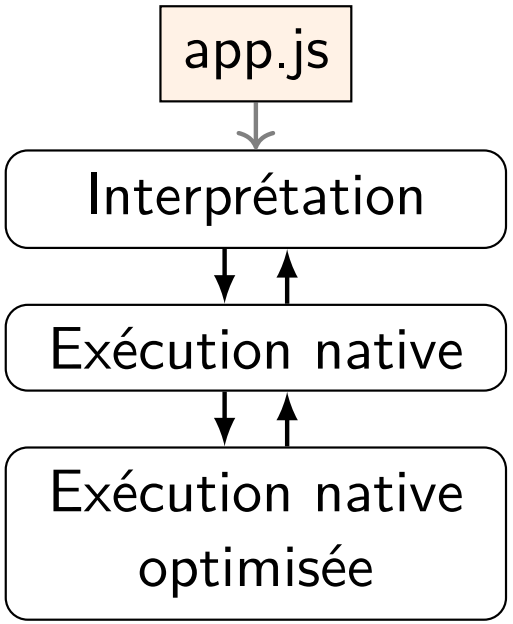


5ème langage - http://pypl.github.io

Exécution JavaScript

Machine virtuelle JavaScript

- Exécution du code en entrée
- Gestion du dynamisme
- Décomposition en couches



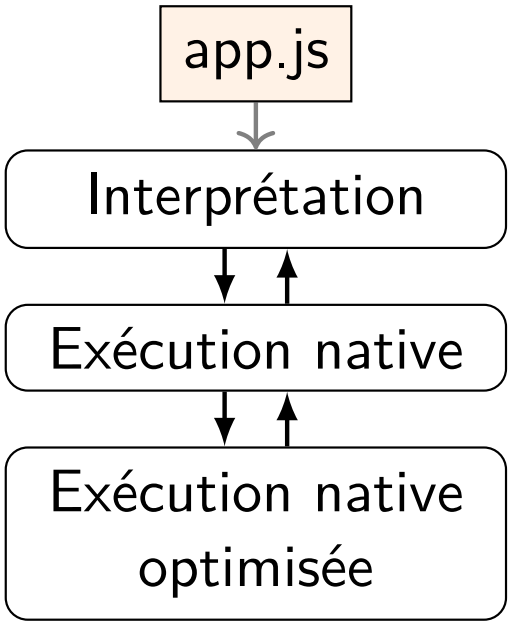
Machines virtuelles commerciales

- SpiderMonkey - *Mozilla*
- JavaScriptCore - *Apple*
- V8 - *Google*
- Chakra - *Microsoft*

Exécution JavaScript

Machine virtuelle JavaScript

- Exécution du code en entrée
- Gestion du dynamisme
- Décomposition en couches



Machines virtuelles commerciales

- SpiderMonkey - *Mozilla*
- JavaScriptCore - *Apple*
- V8 - *Google*
- Chakra - *Microsoft*

Guerre des navigateurs

- Optimisations très poussées
- Néanmoins **sans parallélisme**

Modèle Polyédrique

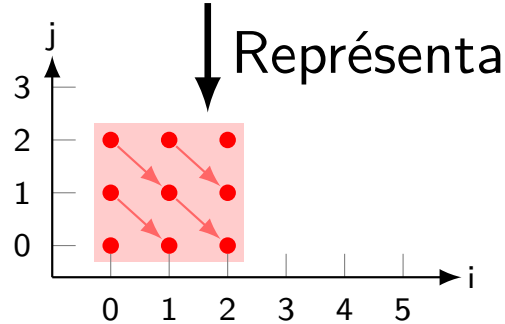
```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    z[i+j] += x[i] * y[j];
```

Modèle Polyédrique

i	j	instances
0	0	z[0] += x[0] * y[0];
0	1	z[1] += x[0] * y[1];
0	2	z[2] += x[0] * y[2];
1	0	z[1] += x[1] * y[0];
1	1	z[2] += x[1] * y[1];
1	2	z[3] += x[1] * y[2];
2	0	z[2] += x[2] * y[0];
2	1	z[3] += x[2] * y[1];
2	2	z[4] += x[2] * y[2];

```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    z[i+j] += x[i] * y[j];
```

Représentation

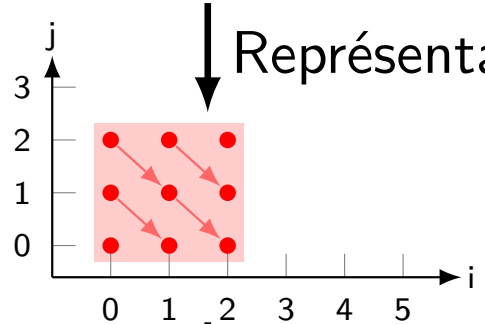


Modèle Polyédrique

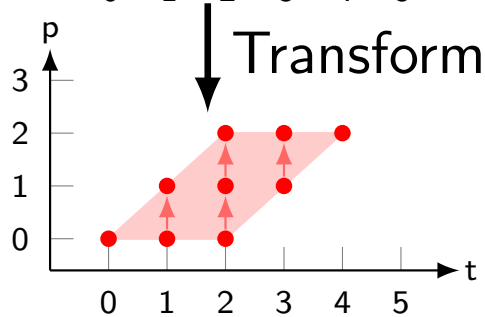
i	j	instances
0	0	z[0] += x[0] * y[0];
0	1	z[1] += x[0] * y[1];
0	2	z[2] += x[0] * y[2];
1	0	z[1] += x[1] * y[0];
1	1	z[2] += x[1] * y[1];
1	2	z[3] += x[1] * y[2];
2	0	z[2] += x[2] * y[0];
2	1	z[3] += x[2] * y[1];
2	2	z[4] += x[2] * y[2];

```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    z[i+j] += x[i] * y[j];
```

Représentation



Transformation: $\begin{pmatrix} t \\ p \end{pmatrix} = \begin{pmatrix} i+j \\ i \end{pmatrix}$

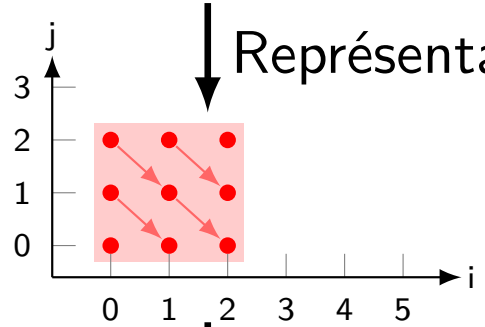


Modèle Polyédrique

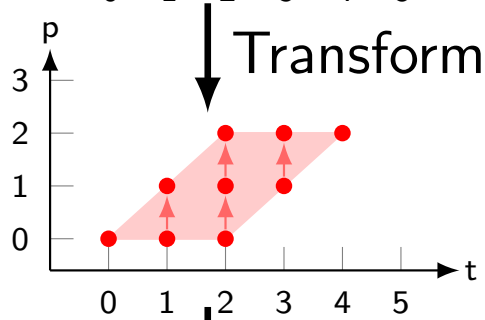
```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    z[i+j] += x[i] * y[j];
```

i	j	instances
0	0	z[0] += x[0] * y[0];
0	1	z[1] += x[0] * y[1];
0	2	z[2] += x[0] * y[2];
1	0	z[1] += x[1] * y[0];
1	1	z[2] += x[1] * y[1];
1	2	z[3] += x[1] * y[2];
2	0	z[2] += x[2] * y[0];
2	1	z[3] += x[2] * y[1];
2	2	z[4] += x[2] * y[2];

Représentation



Transformation: $\binom{t}{p} = \binom{i+j}{i}$

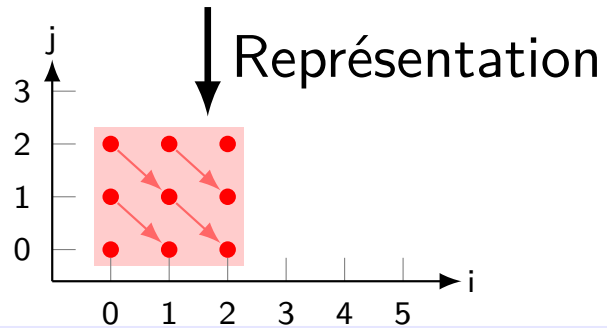


Génération de code

```
#pragma omp parallel for
for (int t = 0; p < 5; p++)
  for (int p = max(0,t-2); p <= min(2,t); p++)
    z[t] += x[p] * y[t-p];
```

Modèle Polyédrique

```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    z[i+j] += x[i] * y[j];
```



Transformations = Optimisations

- Localité des données
- Parallélisation

Static Control Parts (SCoPs)

- Bornes de boucles affines
- Branchements conditionnels affines
- Accès mémoire affines
- Alias connus

Optimisations Polyédriques pour JavaScript

Quelles applications ?

- Traitement images
- Traitement vidéo
- Moteurs de jeux

Optimisations Polyédriques pour JavaScript

Quelles applications ?

- Traitement images
- Traitement vidéo
- Moteurs de jeux

Peut-on faire du polyédrique dans un langage dynamique ?

- SCoPs non détectables statiquement
- Comment et quand détecter les SCoPs ?

Optimisations Polyédriques pour JavaScript

Quelles applications ?

- Traitement images
- Traitement vidéo
- Moteurs de jeux

Peut-on faire du polyédrique dans un langage dynamique ?

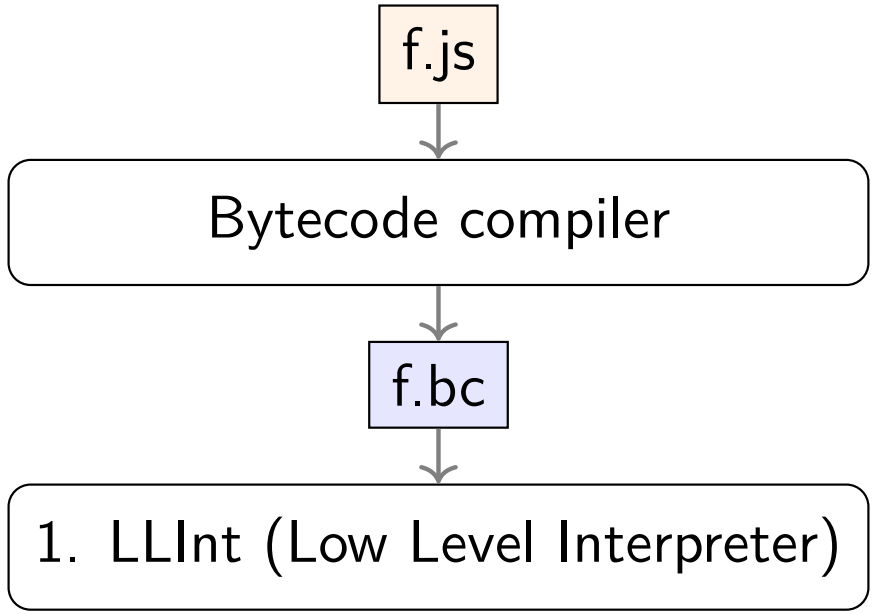
- SCoPs non détectables statiquement
- Comment et quand détecter les SCoPs ?

Est-ce raisonnable de faire du polyédrique au runtime ?

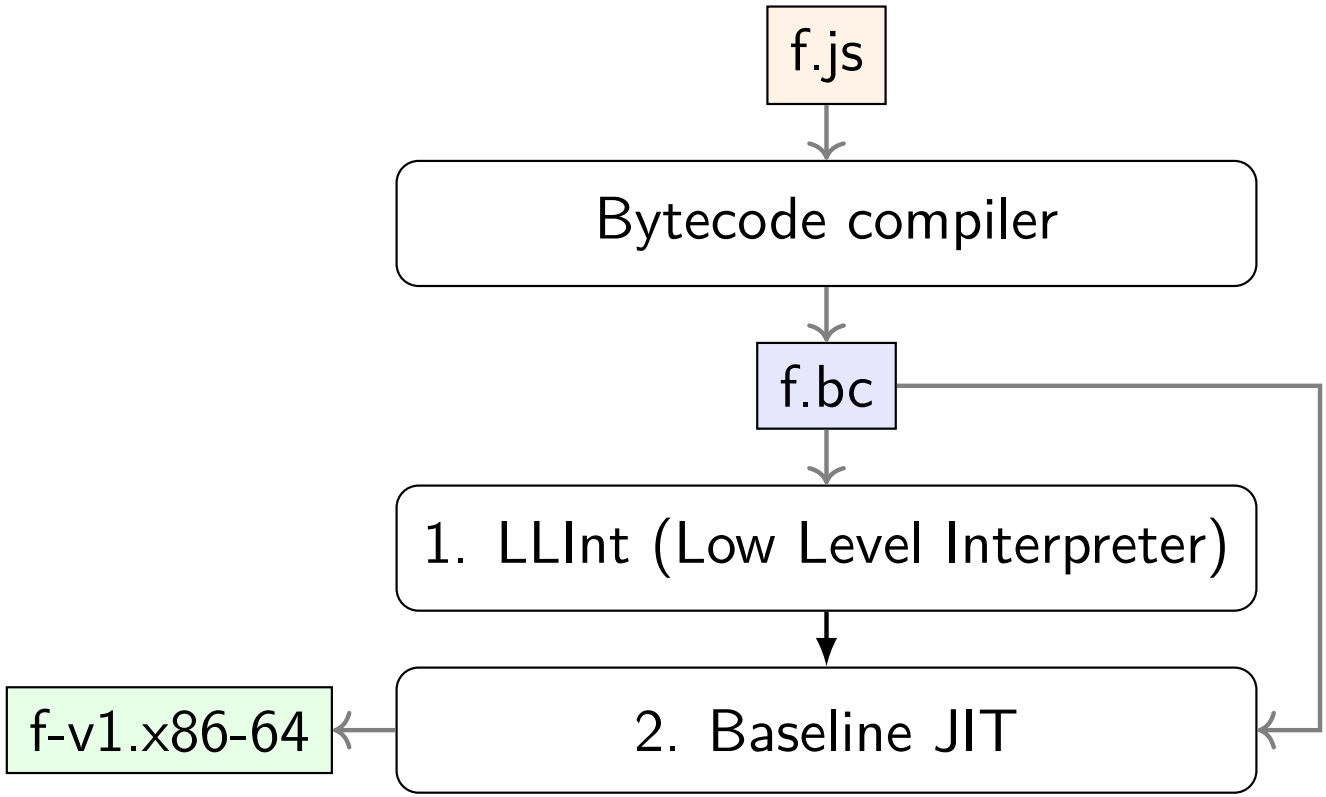
- Compromis gain/surcoût
- APOLLO nous a ouvert la voie

Plan

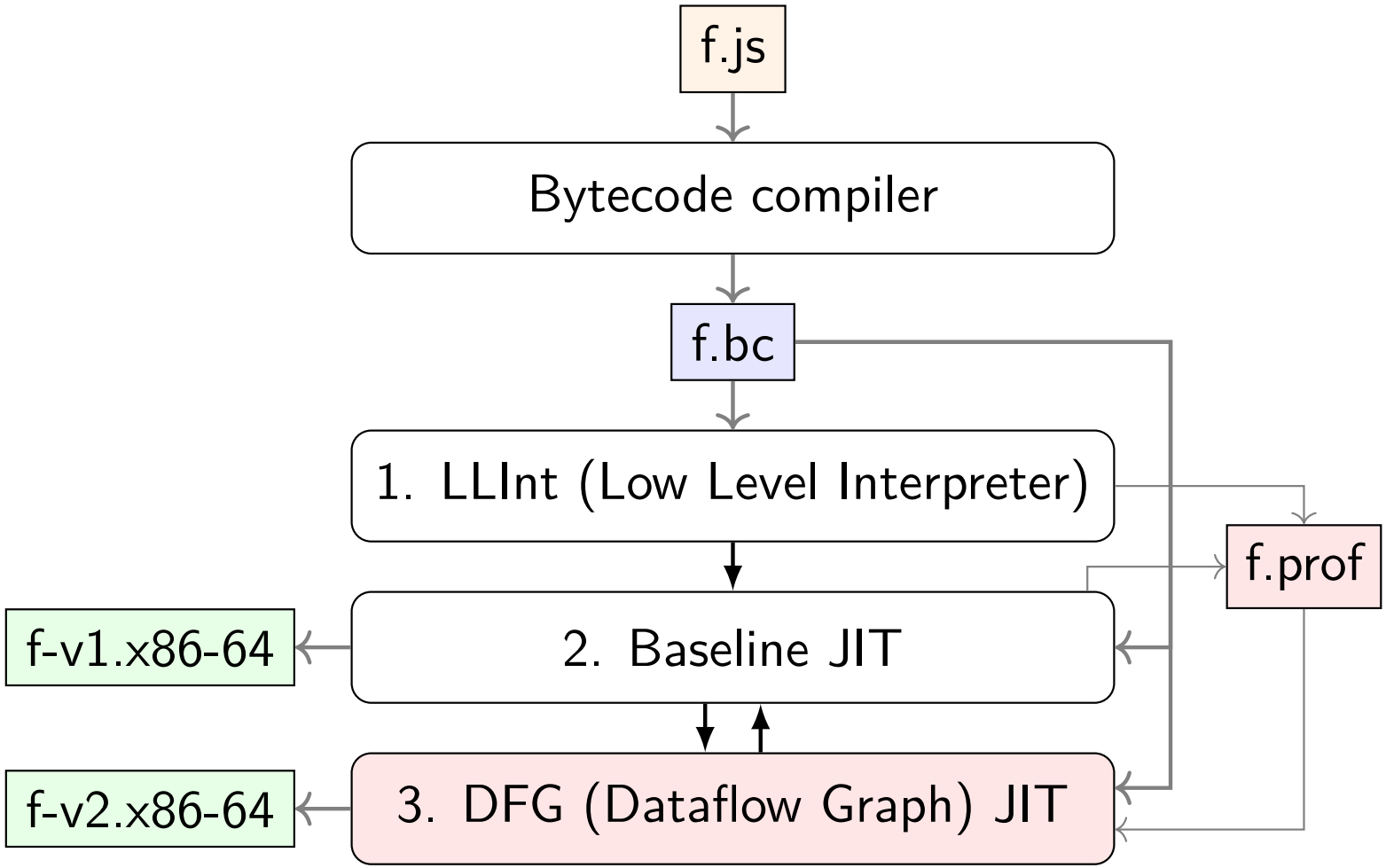
4 Niveaux d'Exécution



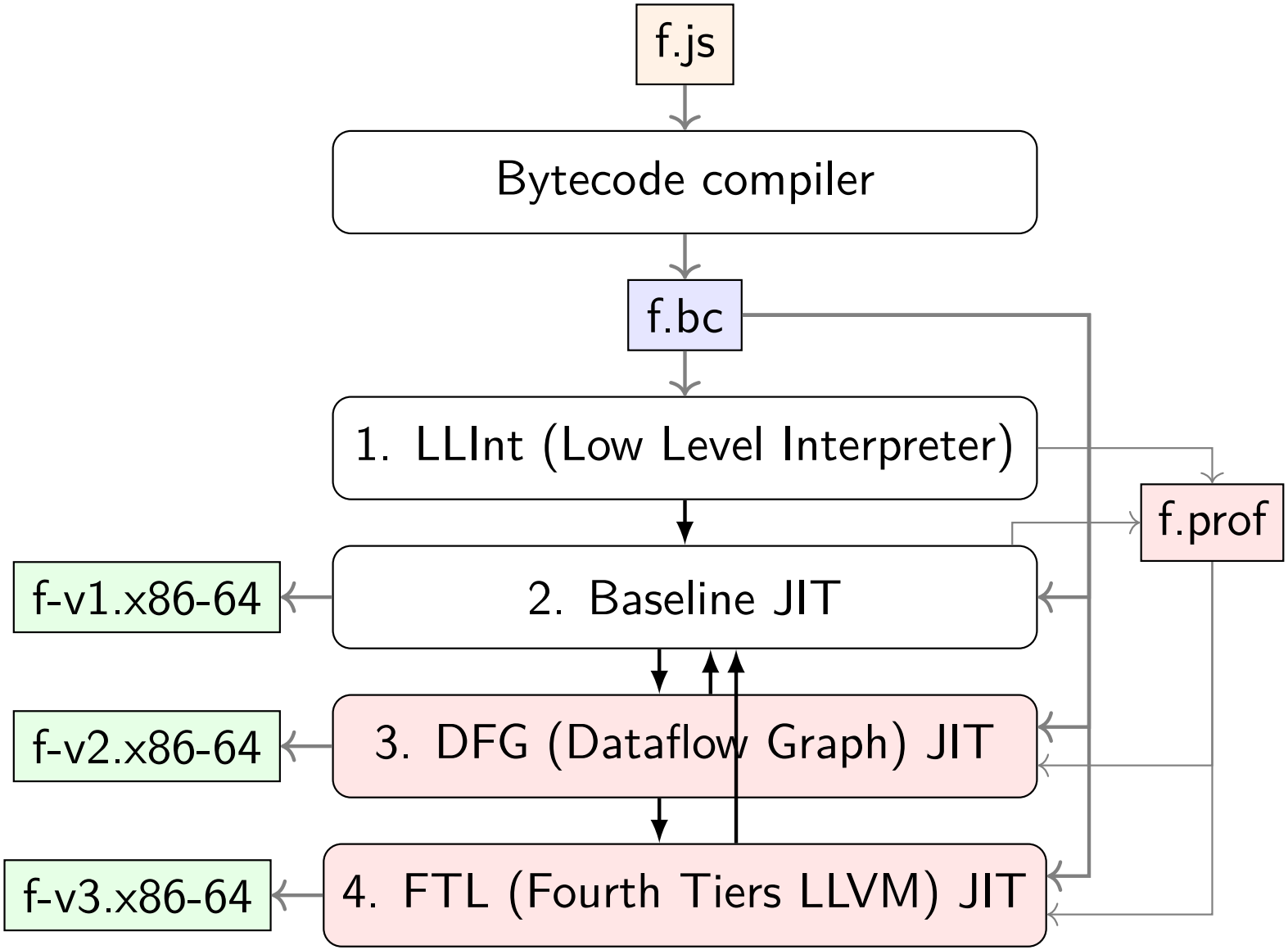
4 Niveaux d'Exécution



4 Niveaux d'Exécution



4 Niveaux d'Exécution



1. LLInt

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

1. LLInt

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

1. LLInt

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

→

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

Interpréteur

```
while(i = next_instruction()) {
  switch(i->opcode) {
    case add:
      switch (typeof(i->operand1->type(), i->operand2->type())):
        case integer_integer:
          i->dest = i->operand1->as_int() + i->operand2->as_int()
        case object_integer:
          ...
    case mul:
      switch (typeof(i->operand1->type(), i->operand2->type())):
        ...
    case get_by_val: ...
  }
}
```

2. Baseline JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

2. Baseline JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

→

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

↓

Code machine

```
...
switch (typeof(v, 41)):
  case integer_integer:
    v = v->as_int() + 41
  case object_integer:
    ...
```

} `op_add v 41 v;`

```
switch (typeof (v, 7)):
  case integer_integer:
    v = v->as_int() * 7
  case object_integer:
    ...
...
```

} `op_mul v 7 v;`

3. DFG JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

→

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

3. DFG JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

→

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

↓

IR DFG typée

```
...
if (v is_integer) {
  v = v->as_int() + 41
  v = v->as_int() * 7
} else {
  return to Baseline JIT
}
...
```

3. DFG JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

→

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

↓

IR DFG typée

```
...
if (v is_integer) {
  v = v->as_int() + 41
  v = v->as_int() * 7
} else {
  return to Baseline JIT
}
...
```

Optim et backend maison

↓

```
...
...
...
```

Code machine

4. FTL JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source


```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

4. FTL JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

↓

IR LLVM typée

```
...
if (v is_integer) {
  v = v->as_int() + 41
  v = v->as_int() * 7
} else {
  return to Baseline JIT
}
...
```

4. FTL JIT

```
f(src_table, dest_table) {
  for (var i = 0; i < 100000; i++) {
    var v = src_table[i]
    v = v + 41;
    v = v * 7;
    dest_table[i] = v;
  }
}
```

Source

```
f(src_table, dest_table) {
  ...
  ...
  op_add v 41 v;
  op_mul v 7 v;
  ...
}
```

Bytecode

IR LLVM typée

```
...
if (v is_integer) {
  v = v->as_int() + 41
  v = v->as_int() * 7
} else {
  return to Baseline JIT
}
...
```

Optim et backend LLVM

```
...
...
...
```

Code machine

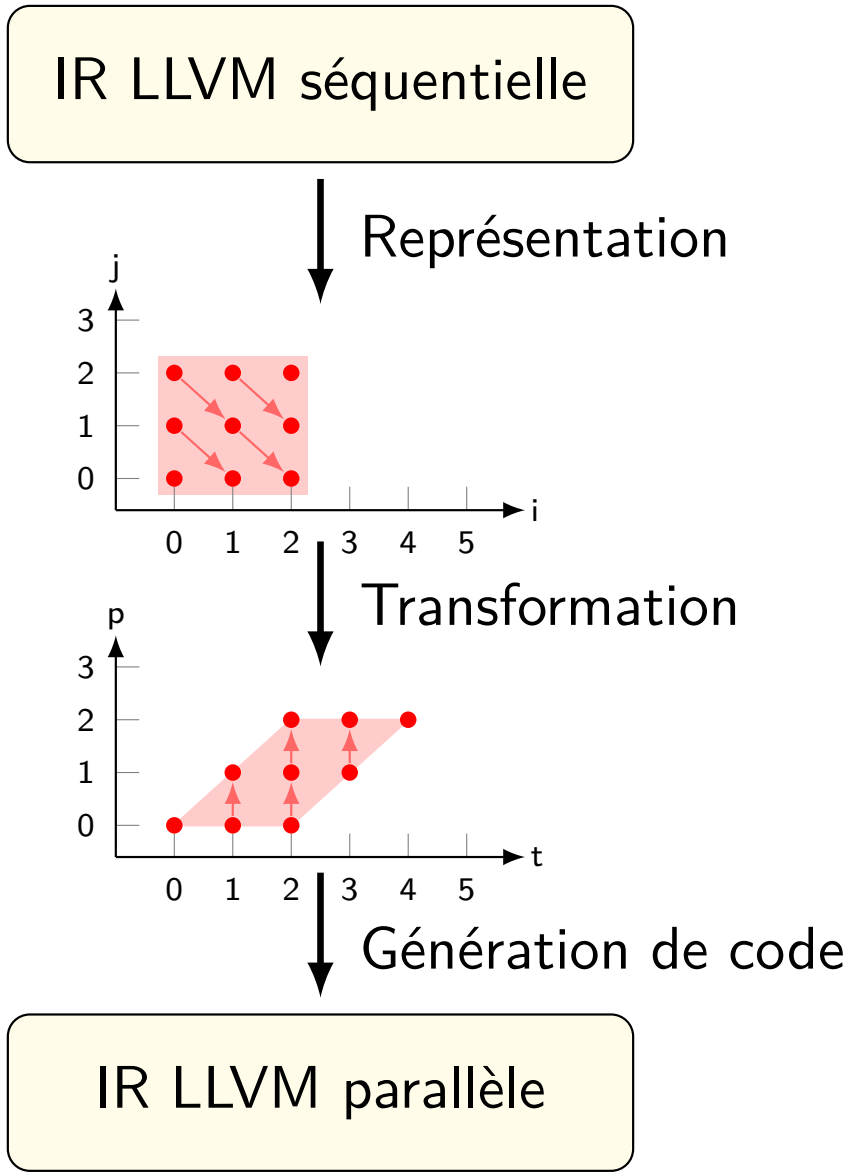
Plan

Objectif

Ajout de transformations polyédriques dans JavaScriptCore

- Dans le dernier niveau (FTL)
- Polly

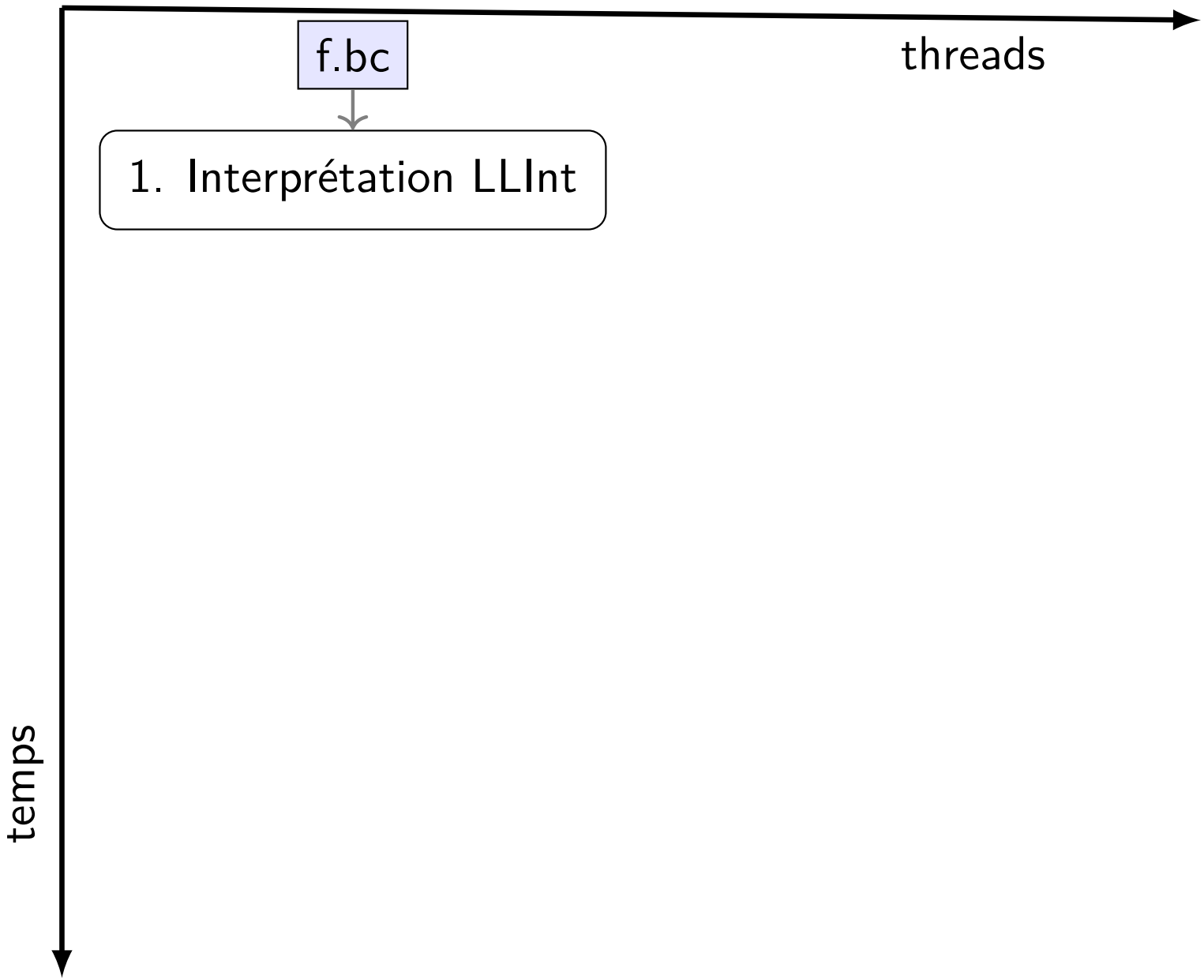
Polly : Optimisations Polyédriques pour LLVM



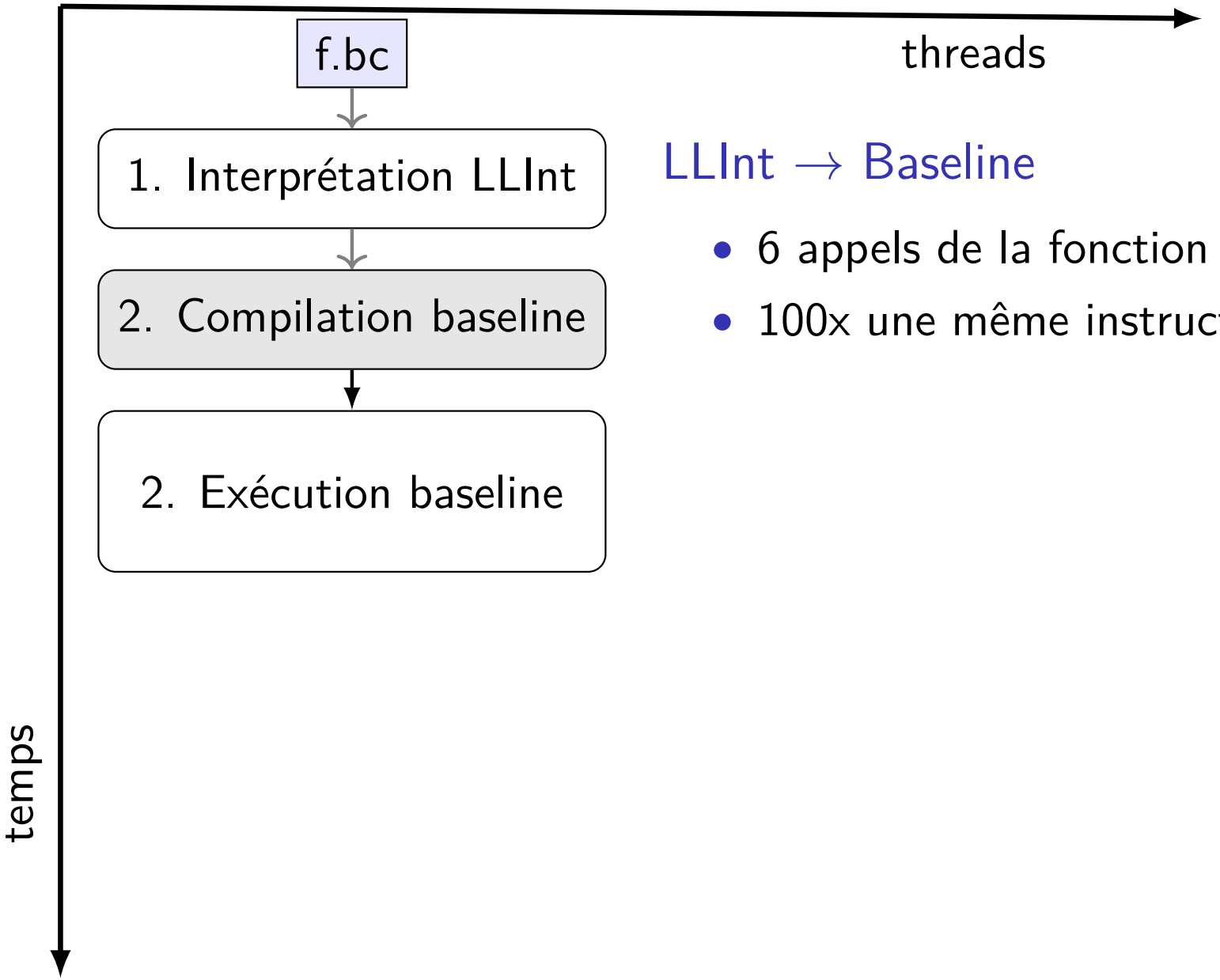
Gain vs Surcoût

Le temps d'exécution de la version optimisée plus le temps d'optimisation de Polly doit être plus court que le temps de la version originale

Réutilisation du Modèle de Coût JavaScriptCore



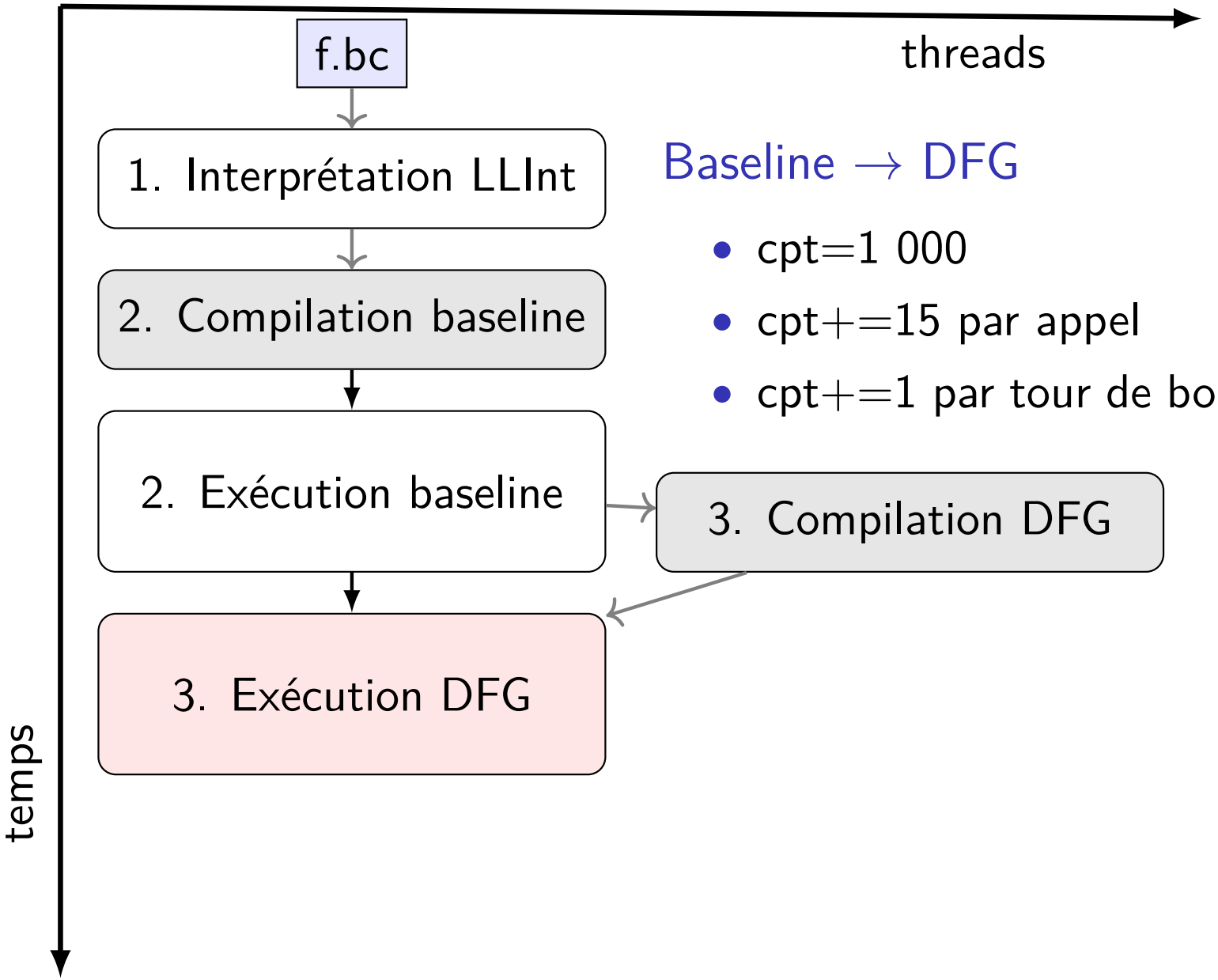
Réutilisation du Modèle de Coût JavaScriptCore



LLInt → Baseline

- 6 appels de la fonction
- 100x une même instruction

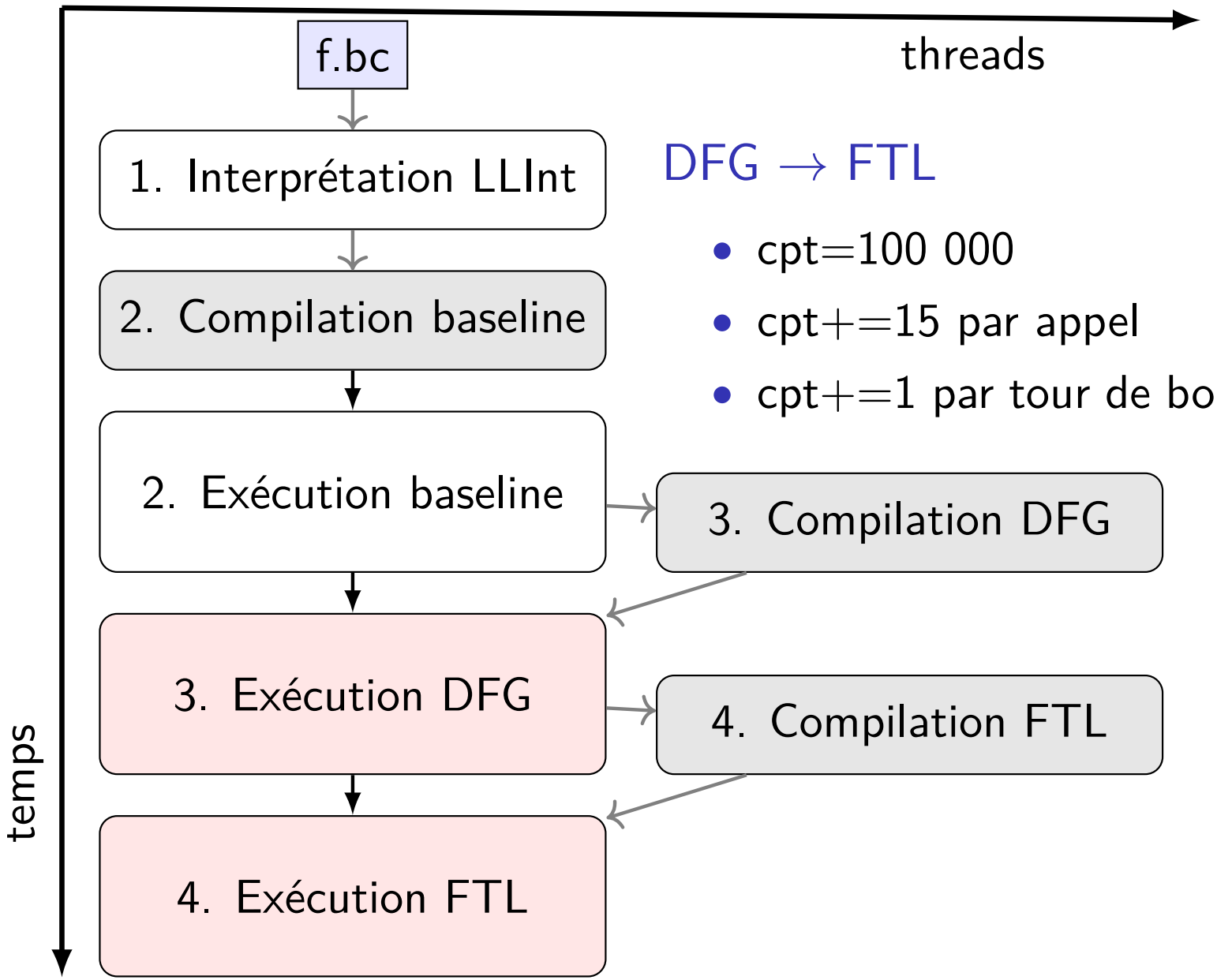
Réutilisation du Modèle de Coût JavaScriptCore



Baseline → DFG

- cpt=1 000
- cpt+=15 par appel
- cpt+=1 par tour de boucle

Réutilisation du Modèle de Coût JavaScriptCore



DFG → FTL

- cpt=100 000
- cpt+=15 par appel
- cpt+=1 par tour de boucle

Plan

Régions SESE : On-Stack Replacement

Mécanisme permettant de changer de niveau

OSR-Entry : niveau suivant

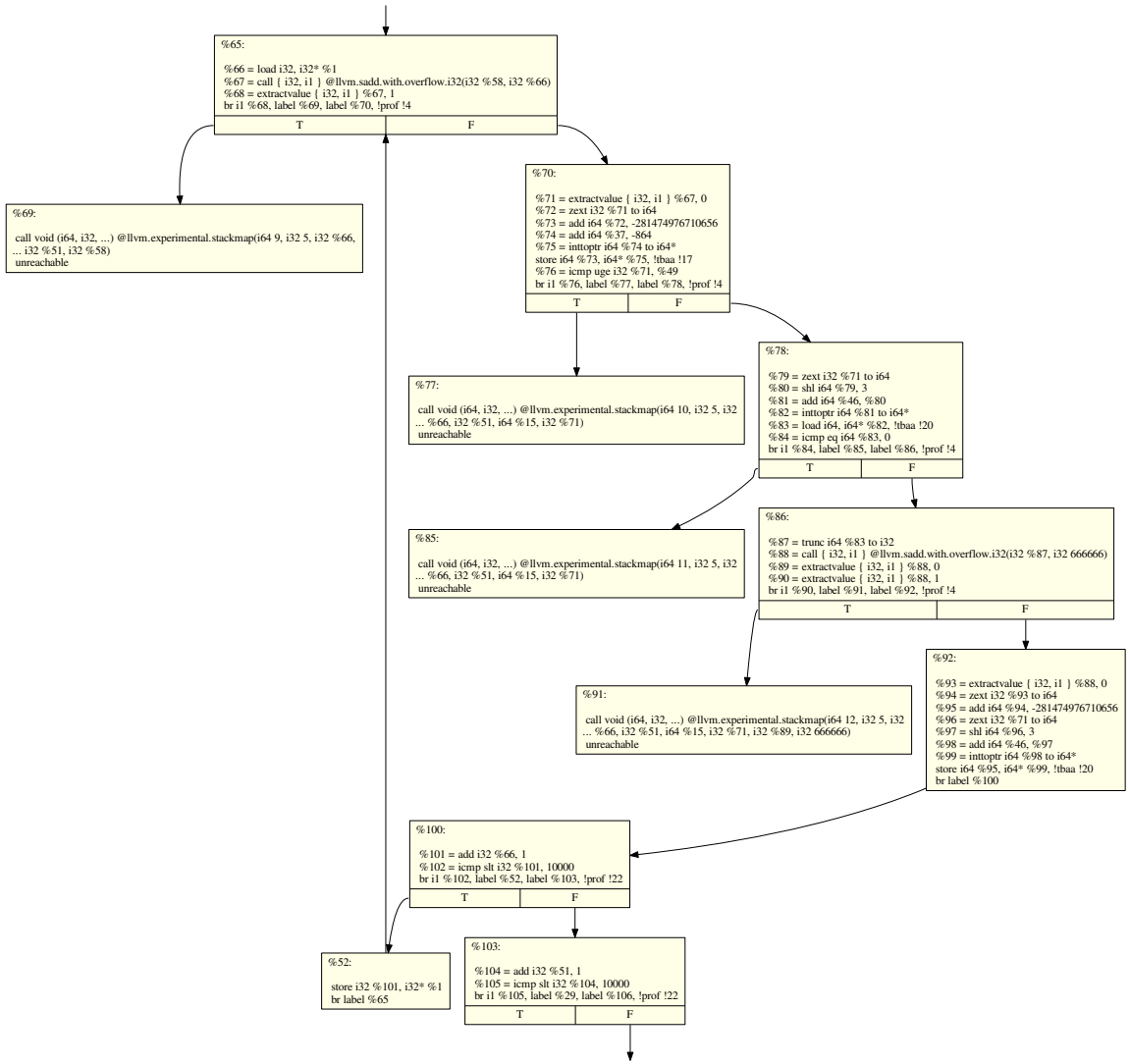
- Boucle exécutée un grand nombre de fois
- Fonction très souvent appelée

OSR-Exit : niveau précédent

- Mauvaise spéculation des types
- Nécessité de réallouer un tableau

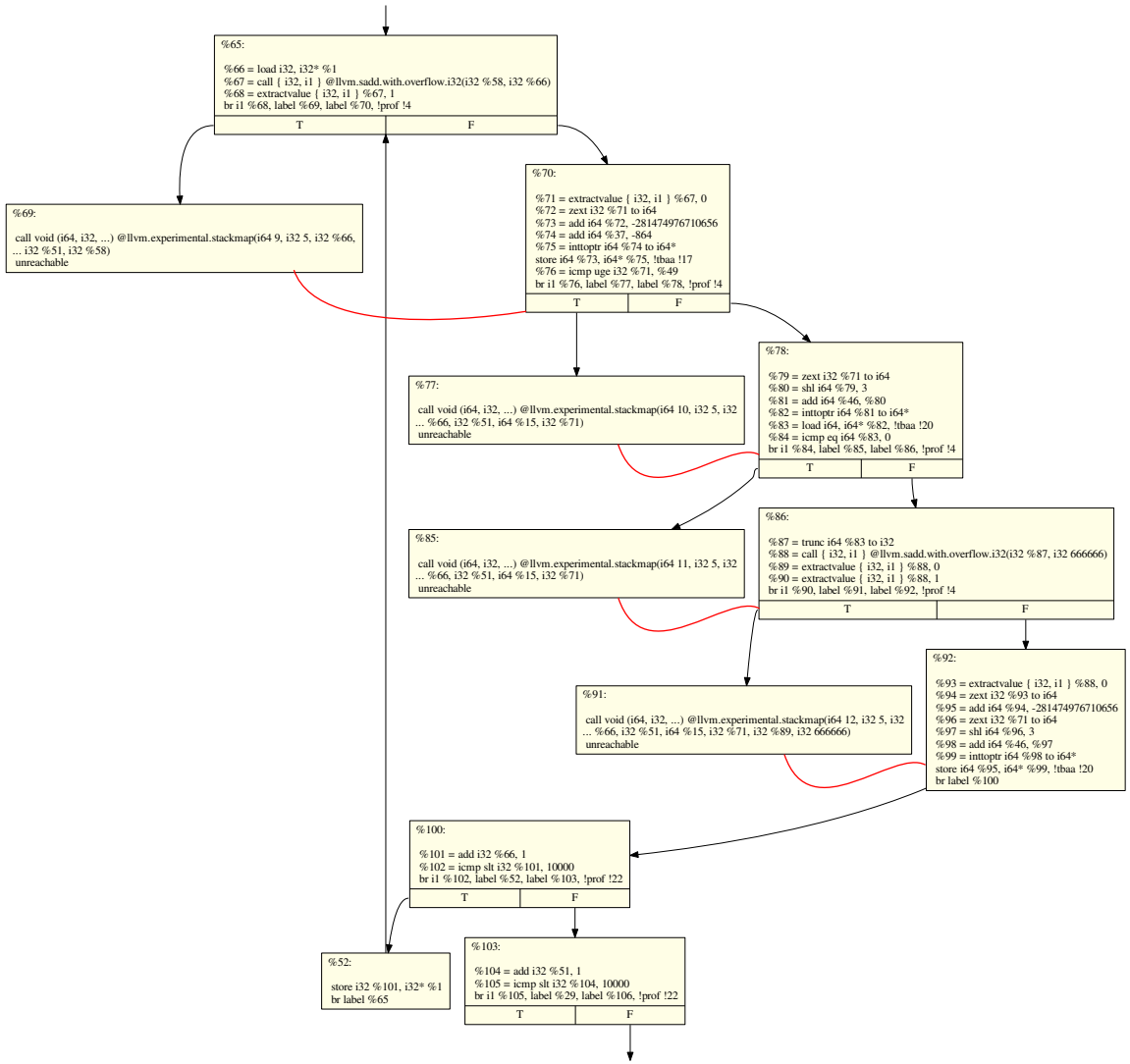
Régions SESE - Problème

Casse le flot de contrôle



Régions SESE : Solution

Raccorder ces blocs au flot "normal"



Aliasing : Tableaux d'Objets

Problème

- Tableau d'objet → deux accès mémoire
  - `t[i].foo = 41`
  - `t[i][j]  = 41`
- Le deuxième accès est fonction du premier
  - Risque d'aliasing
  - On ne peut pas supposer qu'il n'y aura pas d'aliasing statiquement

Solutions possibles

- Ne pas traiter les tableaux d'objets
- Inspecteur-exécuteur : tester l'existence de l'aliasing au runtime

Accès Tableaux : Détection de Fonctions Linéaires

```
t[index] = value;
```

Problème : calculs d'adresse "à la main"

```
%offset = shl i64 %index, 3
%cell_as_int = add i64 %base_as_int, %offset
%cell_ptr = inttoptr i64 %cell_as_int to i64*
store i64 %value, i64* %cell_ptr
```

Accès Tableaux : Détection de Fonctions Linéaires

```
t[index] = value;
```

Problème : calculs d'adresse "à la main"

```
%offset = shl i64 %index, 3
%cell_as_int = add i64 %base_as_int, %offset
%cell_ptr = inttoptr i64 %cell_as_int to i64*
store i64 %value, i64* %cell_ptr
```

Solution : exposition des accès tableaux

```
%base_ptr = inttoptr i64 %base_as_int to [100000 x i64]*
%cell_ptr = getelementptr [100000 x i64],
            [100000 x i64]* %base_ptr,
            i32 0,
            i32 %index
store i64 %value, i64* %cell_ptr
```

Spéculation Parallèle

Problème

- Déclenchement d'un OSR-exit dans un thread parallèle : arrêter l'exécution parallèle

Solutions possibles

- Idempotence : re-exécuter la boucle depuis le début en séquentiel
- Checkpoints : reprendre l'exécution en séquentiel à partir de la sauvegarde

Conclusion

Bilan

- Intégration dans une machine virtuelle existante
- Nids de boucles simples ok
  - Tableaux de primitifs à une dimension

Questions ouvertes

- Quel est le gain **effectif** sur des programmes ?
- Comment exploiter le modèle objet JavaScript pour favoriser la détection d'alias ?
- Comment enrichir le profiling de JavaScriptCore pour aider l'optimiseur polyédrique (à la APOLLO) ?

Merci pour votre attention

Tiers-Up et OSR

LLInt → Baseline

- Représentations de l'état du programme (valeurs des variables) identiques
- OSR entry = jump (rien à copier)
- Entre n'importe quelles instructions de bytecode

Baseline → DFG

- Représentations de l'état du programme différentes
- OSR entry = jump + copie de l'état
- Uniquement à l'entrée de la fonction ou en entête de boucle

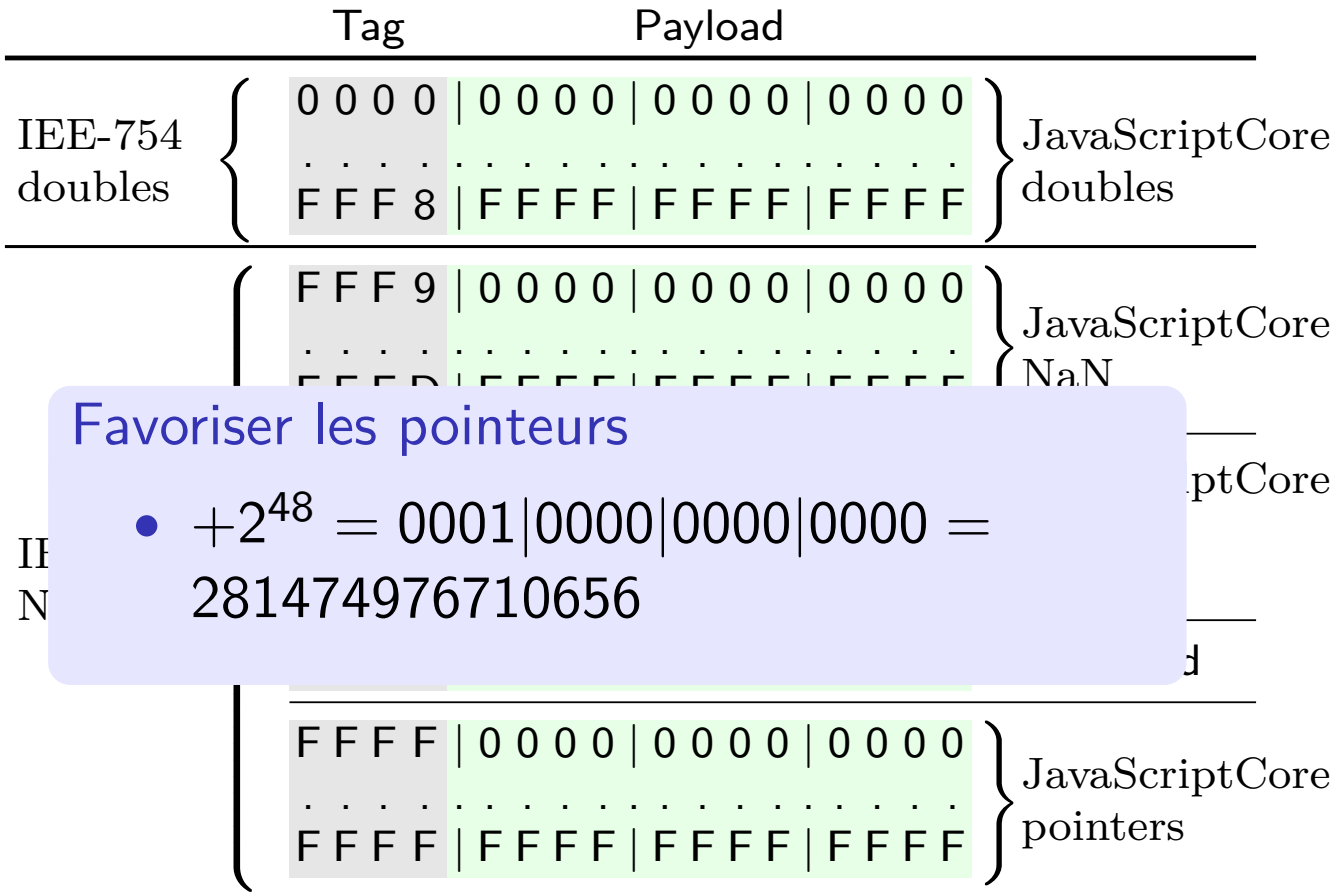
DFG → FTL

- Représentations de l'état du programme différentes
- OSR entry = jump + copie de l'état
- Uniquement à l'entrée de la fonction ou en entête de boucle
    - Deux versions du code

NaN-Boxing dans JavaScriptCore

- Manipulation **directe** des doubles
- Manipulation **indirecte** (masquage) des pointeurs
- Manipulation **indirecte** (masquage) des entiers 32 bits

NaN-Boxing dans JavaScriptCore

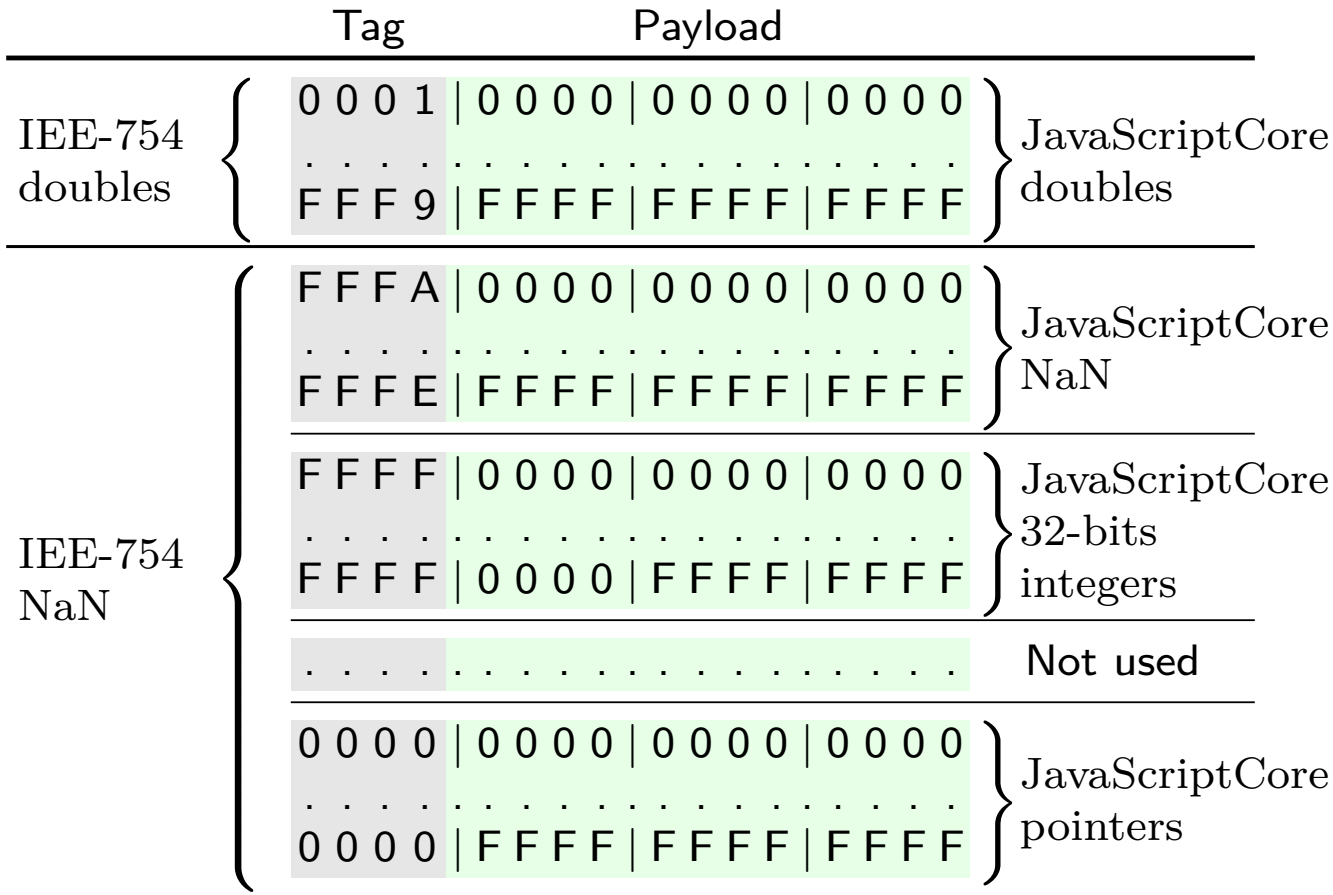


Favoriser les pointeurs

- $+2^{48} = 0001|0000|0000|0000 = 281474976710656$

- Manipulation **directe** des doubles
- Manipulation **indirecte** (masquage) des pointeurs
- Manipulation **indirecte** (masquage) des entiers 32 bits

NaN-Boxing dans JavaScriptCore



- Manipulation **directe** des pointeurs
- Manipulation **indirecte** (soustraction) des doubles
- Manipulation **indirecte** (masquage) des entiers 32 bits

NaN-Boxing dans JavaScriptCore

- `DoubleEncodeOffset`
  - Favoriser les pointeurs
  - $0001|0000|0000|0000 = +281474976710656 = +2^{48}$
- `TagTypeNumber`
  - If all bits in the mask are set, this indicates an integer
  - If any but not all are set this value is a double.
  - $FFFF|0000|0000|0000 = -281474976710656$
- `TagMask = TagTypeNumber | TagBitTypeOther`
  - Used to check for all types of immediate values
  - Either number or other immediate (bool, null, undefined)
  - $FFFF|0000|0000|0002 = -281474976710654$
- add `DoubleEncodeOffset` $\equiv$ sub `TagTypeNumber`

JavaScriptCore

Développé par Apple

- Inclus dans WebKit
- Licence LGPL
- Avant 2008 - SquirrelFish interpréteur
- 2008 - 2014 - Nitro JIT
- 2014 - FTL JIT basé sur LLVM

Gros projet

```
Totaux par langage :
cpp:          289 435  (89.38%)
ansic:         11 254  (3.48%)
ruby:           9 925  (3.06%)
python:         6 195  (1.91%)
asm:            4 982  (1.54%)
perl:           2 013  (0.62%)
sh:                24  (0.01%)
Lignes de code source = 323 828
```