Chapitre 9: Boucles

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251

Où on en est ?

- On sait optimiser du code
- Mais on en veut plus !

Aujourd'hui ? Les boucles

- Le temps est passé dans les boucles
- Un programme sans boucles s'exécute rarement longtemps
- Traitement agnostique des boucles (`for`, `while`, `goto`…)

Difficultés

- Ça tourne !
- Il y a des fonctions affines…

Plan

1. Détection des boucles
2. *Loop invariant code motion* (LICM)
3. Variables d'induction
4. Réduction des variables d'induction
5. Élimination des variables d'induction
6. MiniCC2 -O2
7. Autres optimisations de boucles

Détection des boucles

Boucles naturelles

Arc arrière (*back edge*) dans le CFG

- Arc d'un bloc **dominé** (queue, *latch*)
- À un bloc **dominant** (entête, *header*)
- → Caractérise une boucle

L'entête (*header*)

- Point d'entrée unique dans la boucle
- L'entête **domine** tous les blocs de la boucle

Boucle naturelle

- L'ensemble des blocs dominés par l'entête
- → Pas d'entrée autre que par l'entête
- Sans prédécesseur hors de l'ensemble (sauf pour l'entête bien sur)
- À part avec des `goto` douteux, les boucles des programmes sont toutes naturelles
- → `Loop.java`

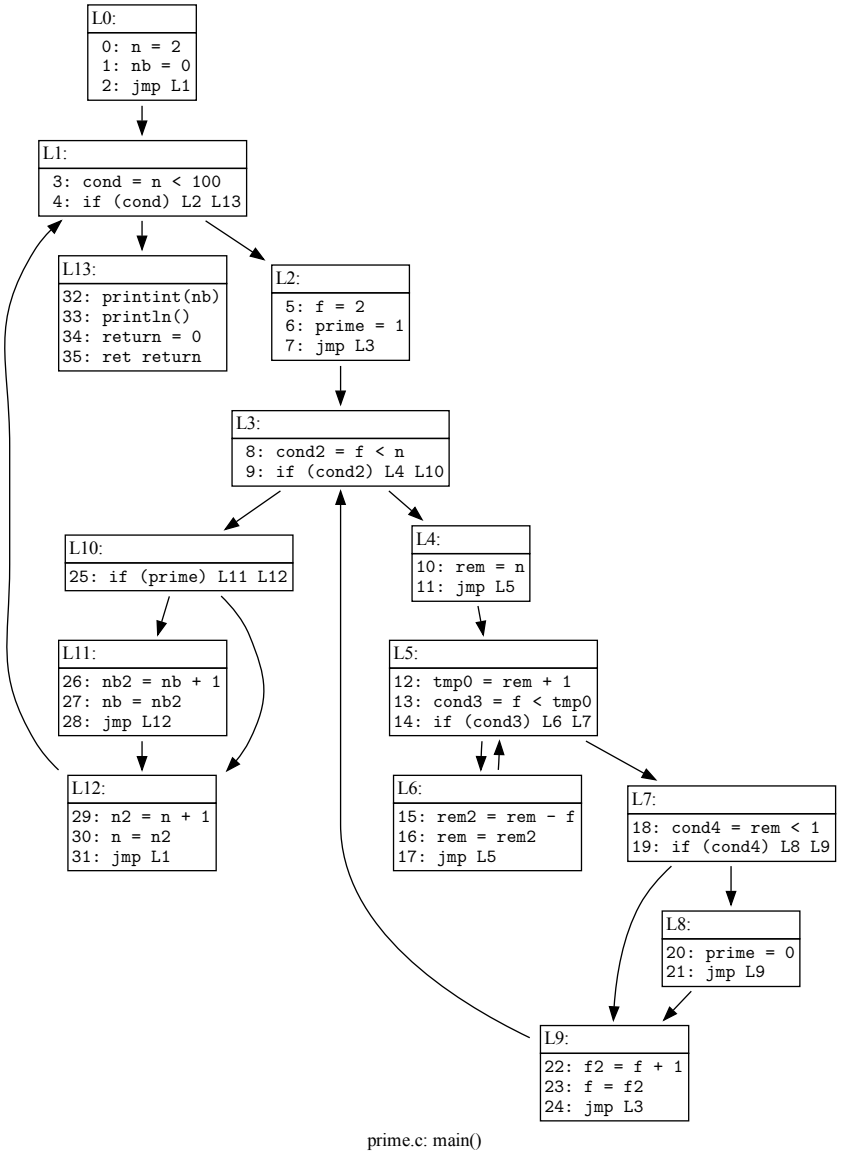
Exercice

- Dessiner une boucle non naturelle (surnaturelle ?)

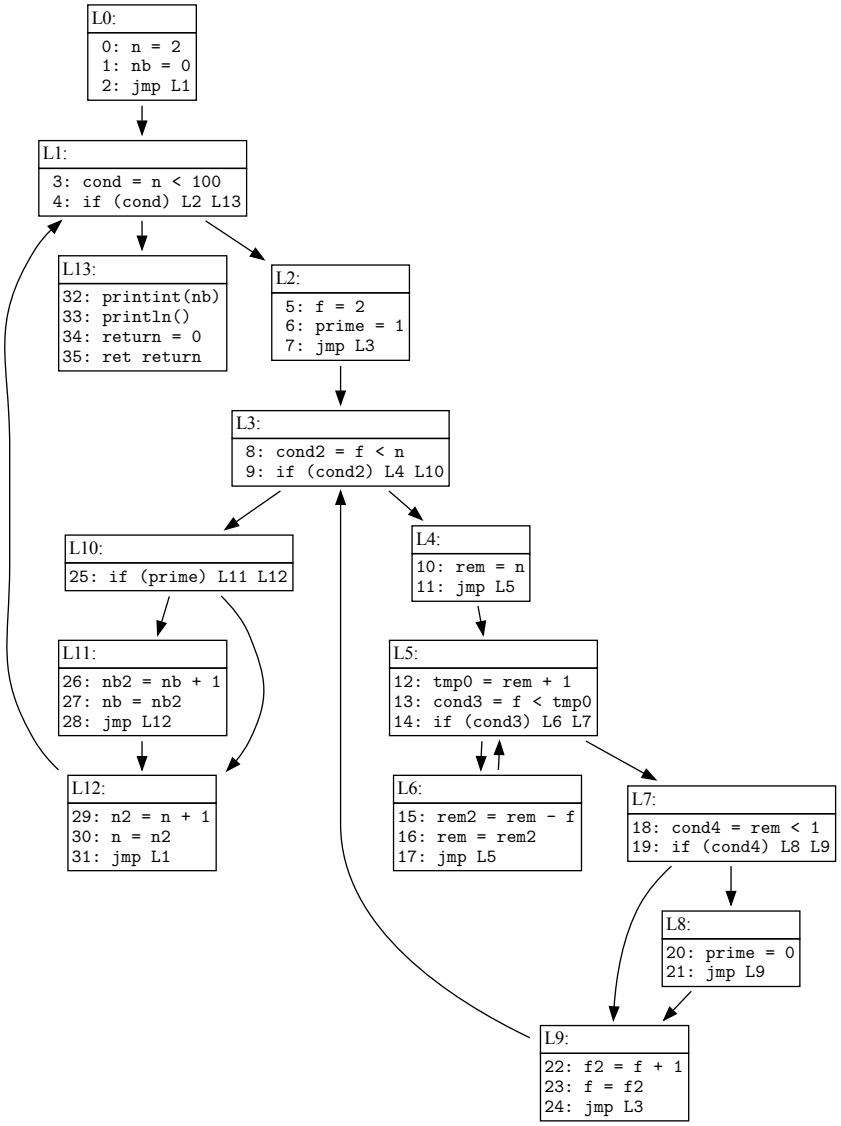
Algorithme : Détection des boucles

- 1: Calculer la relation de dominance (on sait faire)
- 2: Identifier les arcs arrière *queue* $\rightarrow$ *entête* (facile)
- 3: Trouver la boucle associée à chaque arc arrière
  - Supprimer l'*entête* du CFG
  - Collecter tous les nœuds précurseurs de la *queue*
  - $\rightarrow$ Tous les nœuds de l'entête à la queue
- $\rightarrow$ `LoopAnalysis.java`

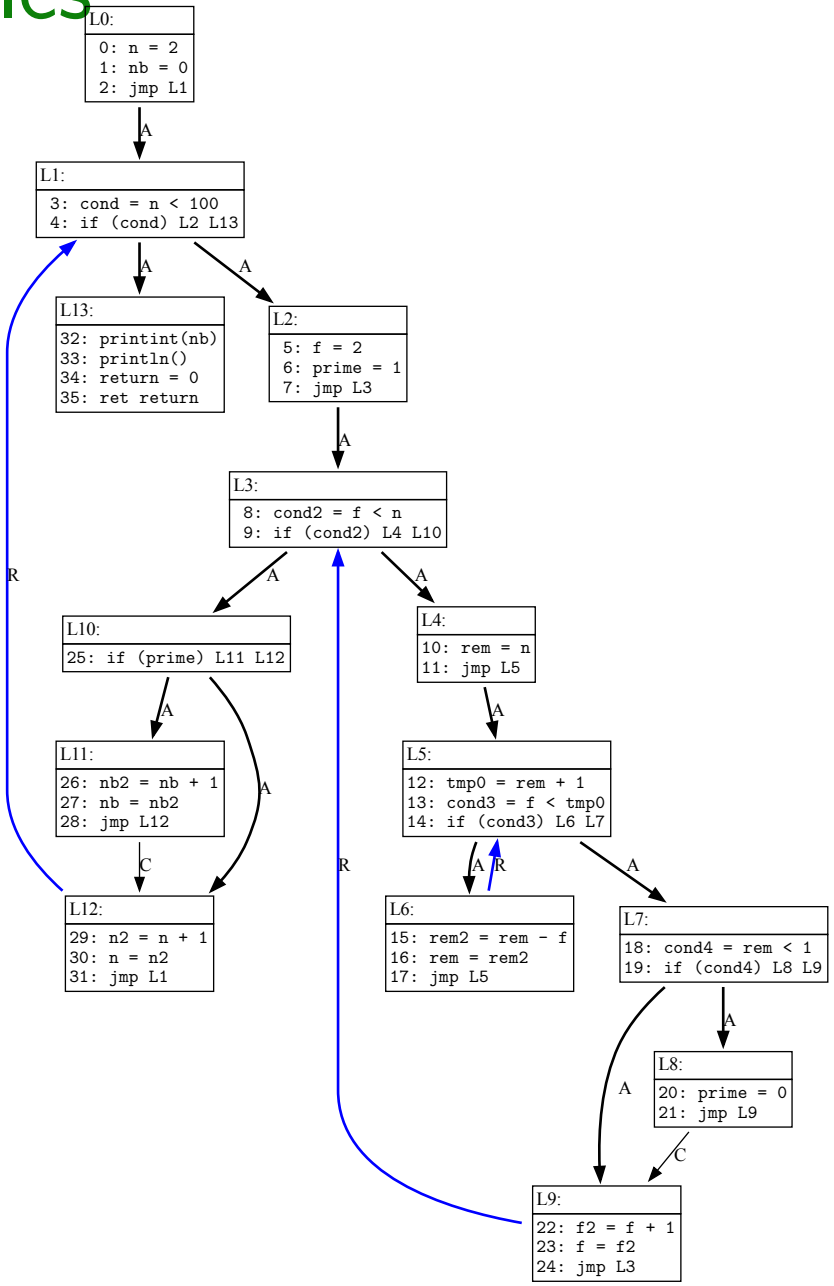
Exercice : Trouver les boucles



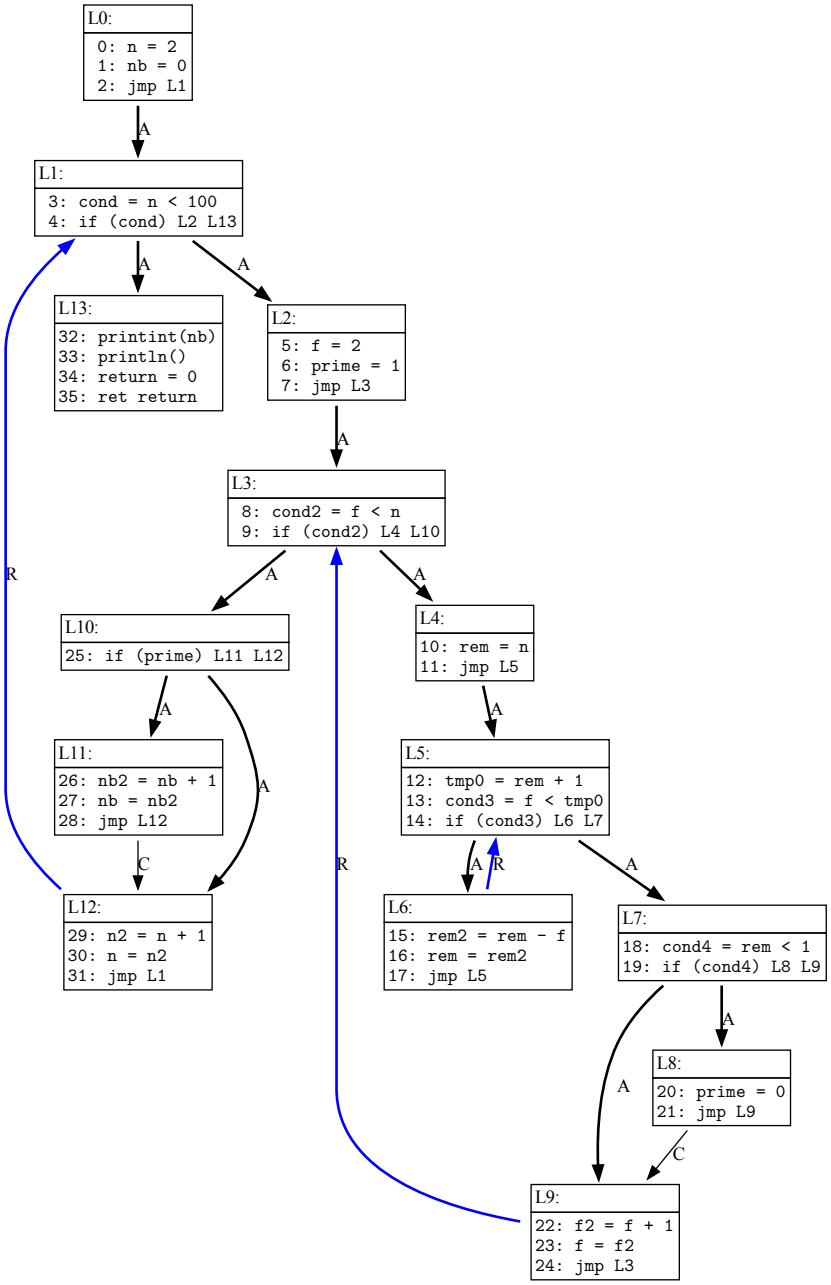
Exercice : Trouver les boucles



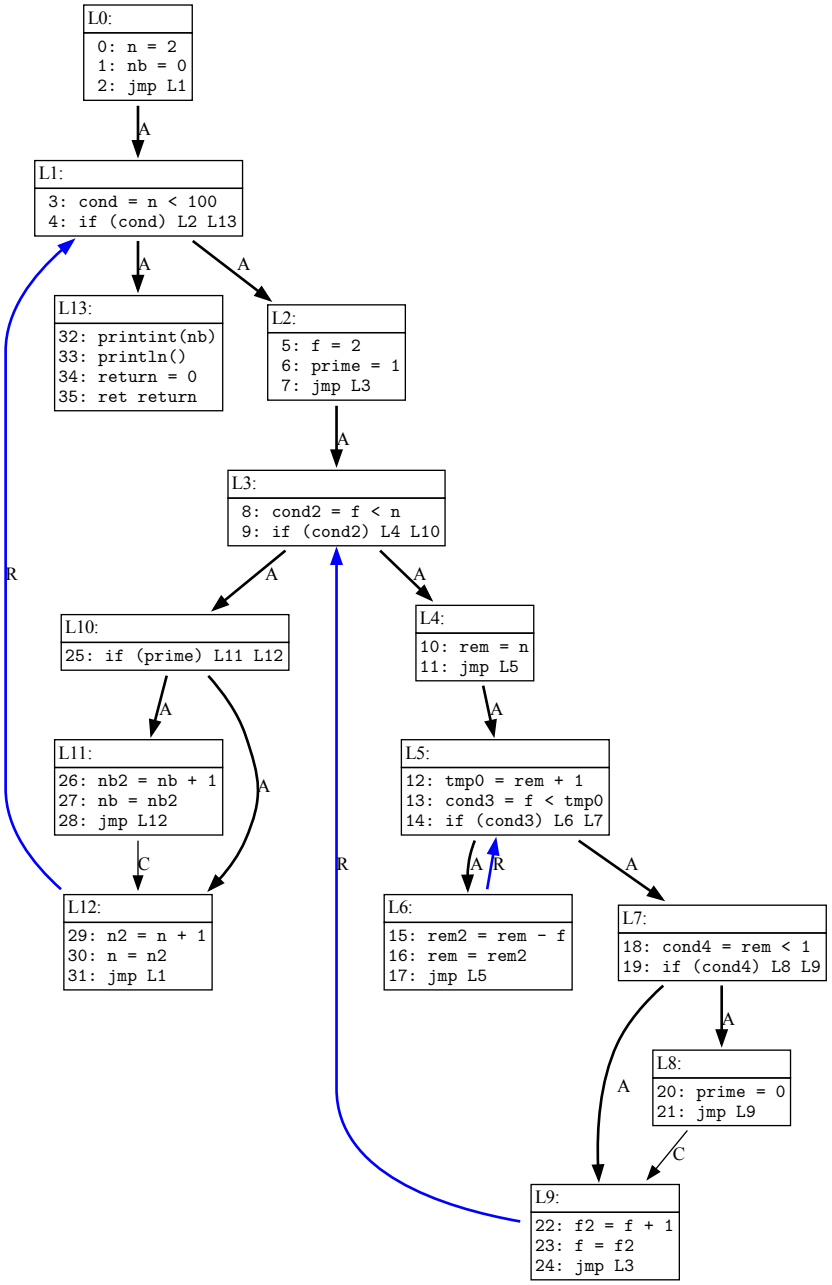
prime.c: main()



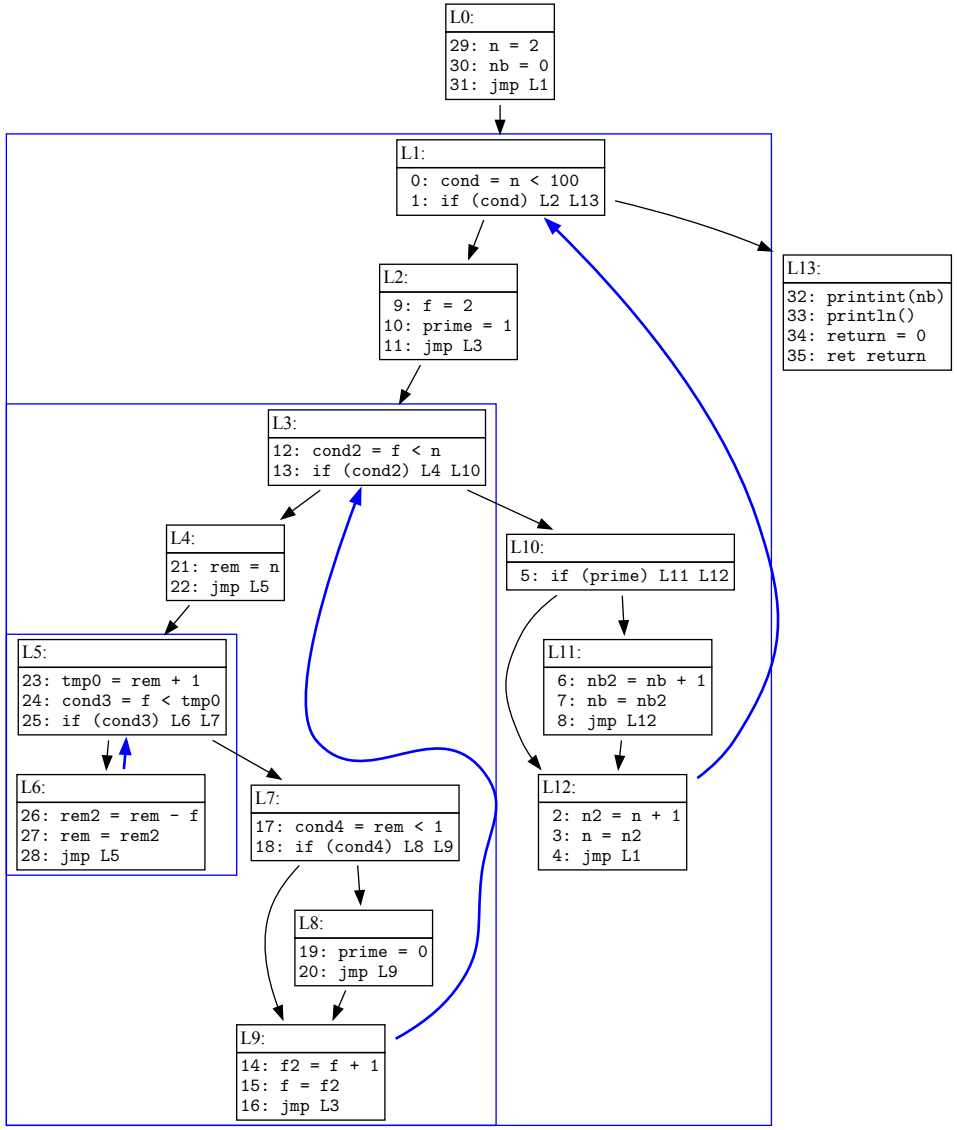
prime.c: main()


L0:
0: n = 2
1: nb = 0
2: jmp L1
A
L1:
3: cond = n < 100
4: if (cond) L2 L13
A
A
L13:
32: printint(nb)
33: println()
34: return = 0
35: ret return
L2:
5: f = 2
6: prime = 1
7: jmp L3
A
L3:
8: cond2 = f < n
9: if (cond2) L4 L10
A
A
R
L10:
25: if (prime) L11 L12
A
L11:
26: nb2 = nb + 1
27: nb = nb2
28: jmp L12
A
C
L12:
29: n2 = n + 1
30: n = n2
31: jmp L1
R
L4:
10: rem = n
11: jmp L5
A
L5:
12: tmp0 = rem + 1
13: cond3 = f < tmp0
14: if (cond3) L6 L7
A
R
A
L6:
15: rem2 = rem - f
16: rem = rem2
17: jmp L5
L7:
18: cond4 = rem < 1
19: if (cond4) L8 L9
A
A
L8:
20: prime = 0
21: jmp L9
C
L9:
22: f2 = f + 1
23: f = f2
24: jmp L3

prime.c: main()



prime.c: main()



prime.c: main()

Boucles imbriquées

Boucle imbriquée

- Une boucle totalement **incluse** dans une autre
- Boucle intérieure (*inner loop*): une boucle sans boucle imbriquée
- $\rightarrow$ Arborescence de boucles

Configurations possibles

- Si entêtes différentes
  - Boucles imbriquées
  - ou boucles entièrement disjointes
- Si entête commune
  - Traiter comme une boucle unique
  - $\rightarrow$ Ajouter une queue commune si on souhaite garder un seul arc arrière
  - Exercice : trouver un exemple

Blocs auxiliaires

- Pour plusieurs transformations, c'est utile d'avoir des blocs particuliers supplémentaires
- Les trouver, ou les créer si besoin

Pré-entête (*pre-header*)

- Bloc prédécesseur de l'entête
- Qui n'est pas dans la boucle
- Qui est unique
- Qui n'as pas d'autres successeurs que l'entête
- $\rightarrow$ La pré-entête domine la boucle sans en faire partie
- $\rightarrow$ L'entête post-domine la pré-entête

Intérêt

- Distinguer facilement ce qui vient de dehors et de dedans
- Mettre des traitements qui ont lieu juste avant d'entrer

Blocs auxiliaires

Queue unique

- En cas de queues multiples, fusionner
- $\rightarrow$ Un seul arc arrière
- Mettre des traitements qui ont lieu entre deux tours

Sorties dédiées (*dedicated exits*)

- Blocs successeurs de la boucle
- Qui ne sont pas dans la boucle
- Qui sont dominés par l'entête
- $\rightarrow$ Les sorties sont exclusives à la boucle
- Intérêt : mettre des traitements qui ont lieu juste à la sortie

Analyse de coût (*cost analysis*)

- Évaluer la performance d'un CFG **à la louche**
- Compter le nombre d'instruction !

Analyse de coût (*cost analysis*)

- Évaluer la performance d'un CFG **à la louche**
- Compter le nombre d'instruction !
- $\rightarrow$ Pas mal, mais les boucles ?

Analyse de coût (*cost analysis*)

- Évaluer la performance d'un CFG **à la louche**
- Compter le nombre d'instruction !
- → Pas mal, mais les boucles ?
- Considérer que chaque boucle est exécutée 10 fois
- → `CostAnalysis.java`

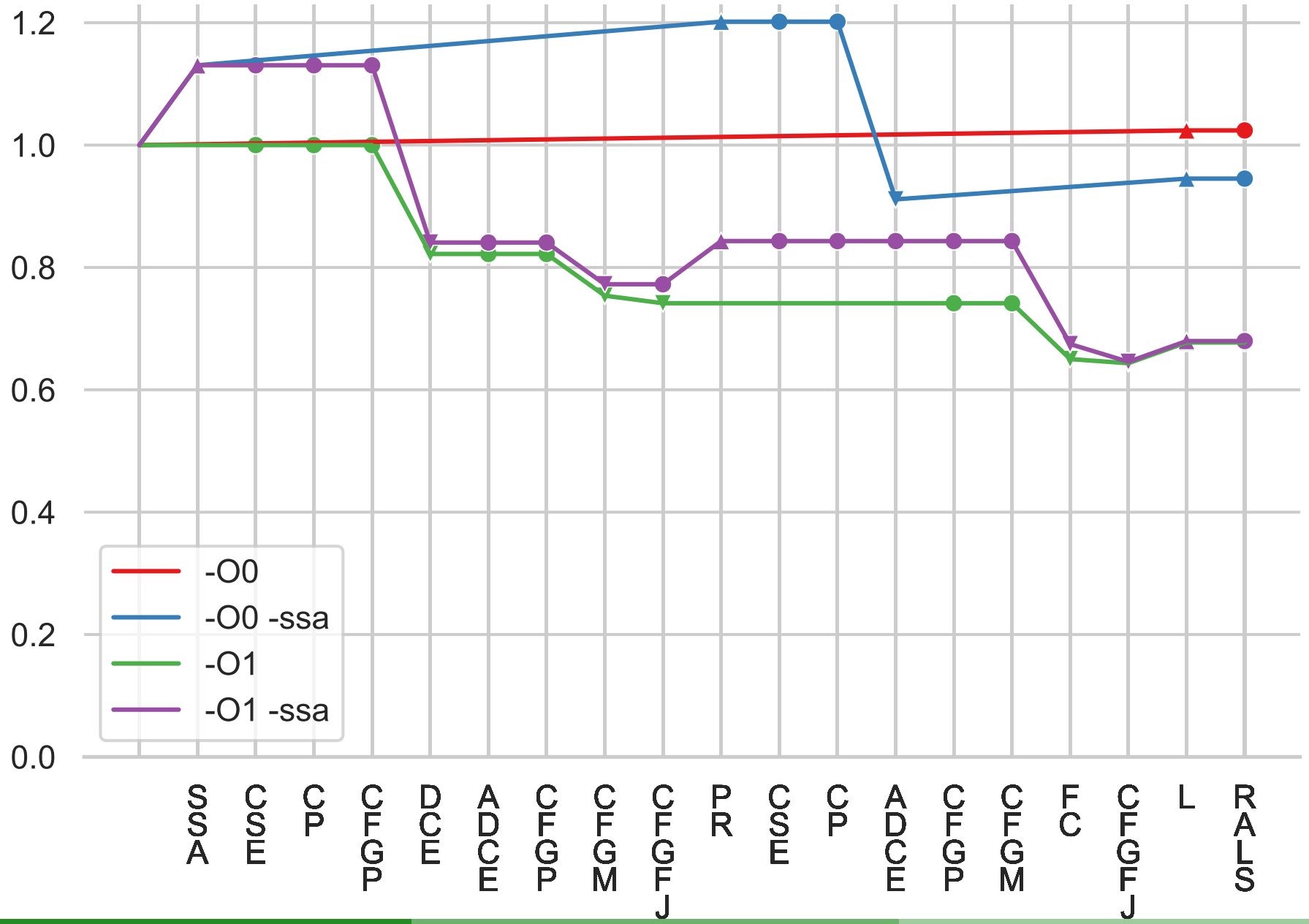
Pourquoi ?

- Évaluer **à la louche** l'impact du *spill* d'un registre
- Évaluer **à la louche** l'impact d'une transformation

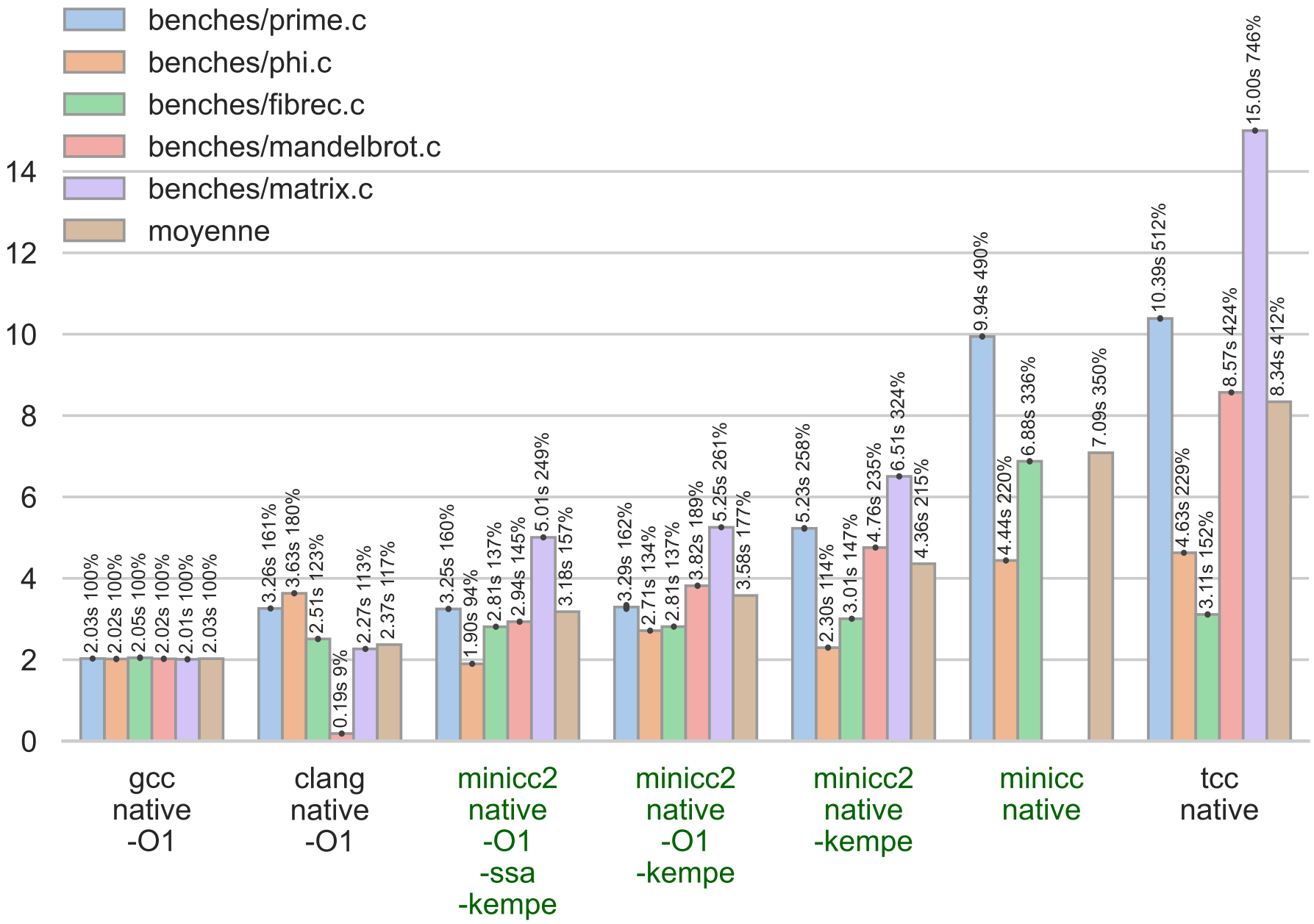
On peut faire mieux

- Estimer le coût des instructions individuelles
- Estimer le coût collectif des instructions (pipelines et caches)
- Estimer la probabilité d'atteindre un bloc
- Estimer plus précisément le nombre de tours de boucles
- Comparer avec des coûts mesurés dans la vrai vie

Impact des passes MiniCC2, à la louche



Mesures métal



Loop invariant code motion (LICM)

Loop invariant code motion (LICM)

Invariant de boucle

- Une instruction sans effet de bord
- Qui dépend seulement de registres définis hors boucle
  $\rightarrow$ Ou de registres eux-mêmes invariants de boucle
- $\rightarrow$ Garde la même valeur en entrée, pendant et en sortie de boucle

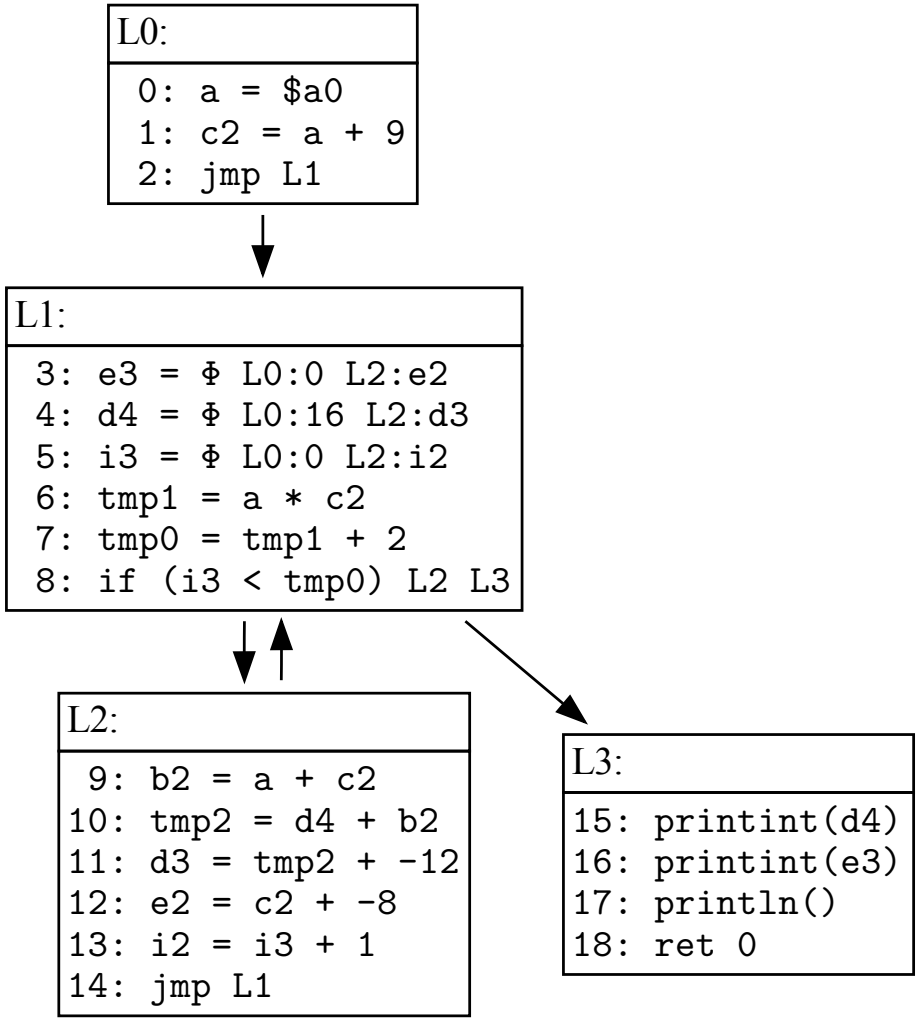
Remontée (*code hoisting*)

- Peut-être déplacée dans le bloc de pré-entête
- $\rightarrow$ `LoopInvariantCodeMotion.java`
- Gains ? Pertes ?

Sans SSA

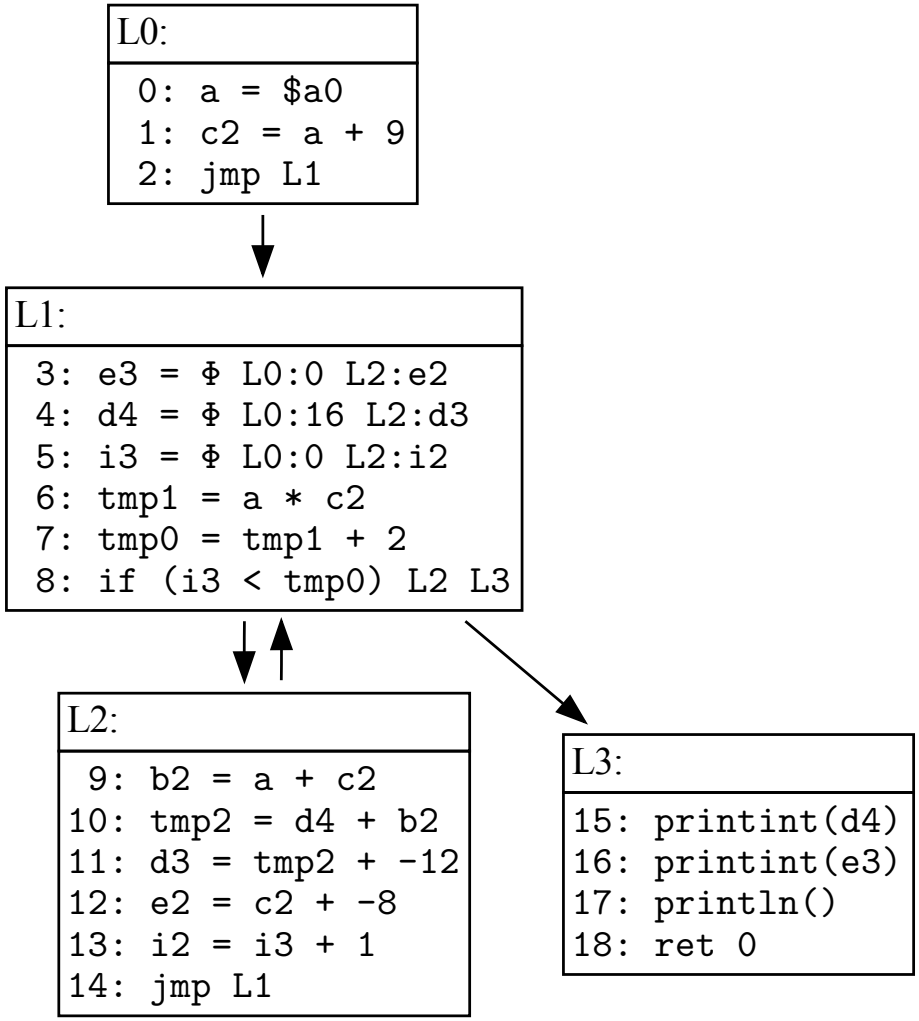
- Les règles de détection sont un peu plus complexes
- Il y a des contraintes de transformations
- $\rightarrow$ Avec SSA c'est plus facile !

Exercice : Remonter les invariants de boucle

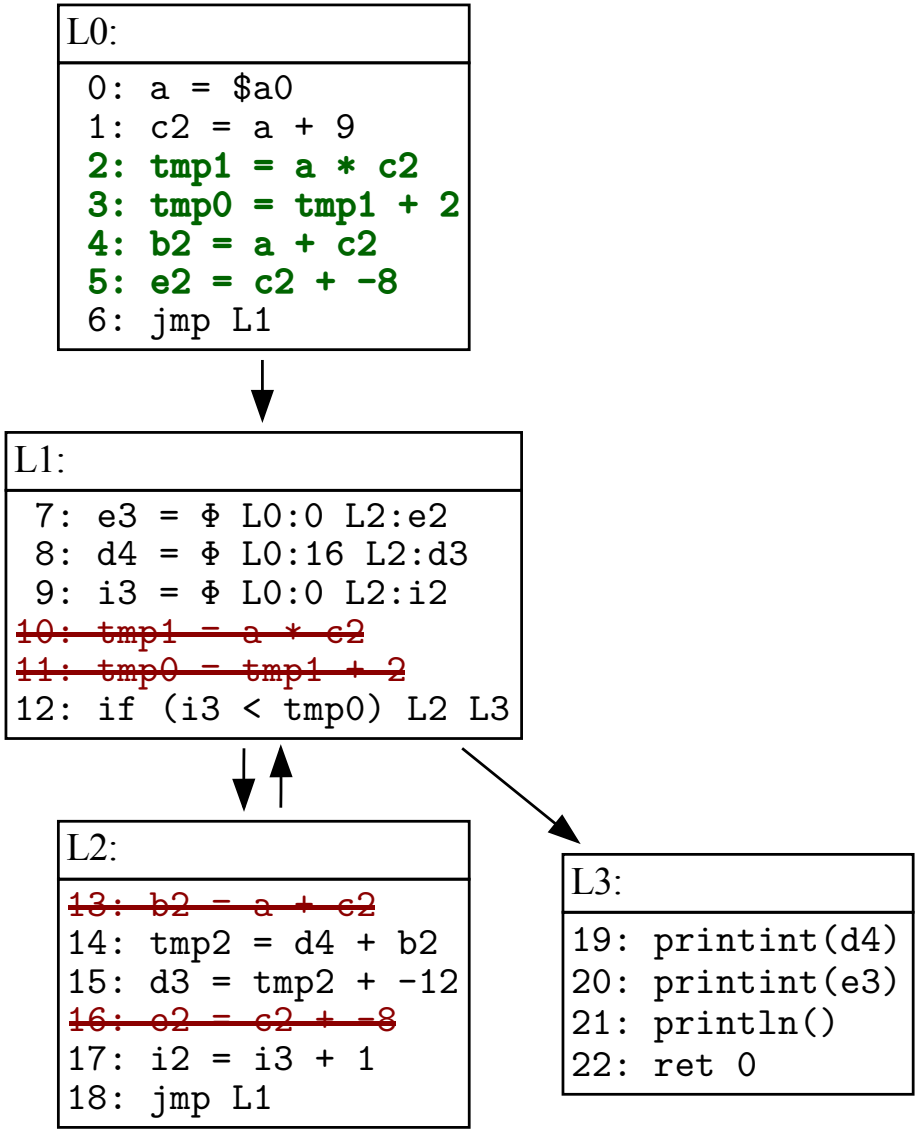


licm.c: foo()

Exercice : Remonter les invariants de boucle



licm.c: foo()



licm.c: foo()

Variables d'induction

Variable d'induction

On ne considère donc **que** les blocs d'une boucle

Variable d'induction basique `i`

- La **seule** instruction affectant `i` dans la boucle est de la forme
- `i = i + ic` (ou `i = i - ic`)
- où `ic` est un invariant de boucle (constante ou variable)
- Note: `i` a une valeur initiale (aussi un invariant de boucle)
- → `BasicInductionVariable.java`

En SSA, il faut deux instructions :(

- Et pas nécessairement dans le même bloc de base
- `i1 = Φ(LA:i0 LB:i2); ...; i2 = i1 + ic;`
- `i1` est la valeur avant l'incrément
- `i0` est la valeur initiale (invariant)
- `i2` est la valeur après l'incrément
- `ic` est l'incrément (invariant)

Variable d'induction dérivée

Variable d'induction dérivée `k`

- La seule instruction affectant `k` dans la boucle est `k = j @ ko`
- où `j` est une variable d'induction (basique ou dérivée)
- où `@` est l'opération `+`, `-` ou `*` (et variation commutative)
- où `ko` est un invariant de boucle (l'opérande)
- → `DerivedInductionVariable.java`

Détection

- En SSA, c'est facile à trouver !
- → `InductionVariableDetection.java`

Dérivée vs. basique

- `j` dérivée de `i` : `j = i @ jo`
- Quelle est la relation mathématique entre i et j ?

Dérivée vs. basique

- `j` dérivée de `i` : `j = i @ jo`
- Quelle est la relation mathématique entre i et j ?
  - $j = i + jc = 1 \cdot i + jc$
  - $j = i - jc = 1 \cdot i + -jc$
  - $j = jc - i = -1 \cdot i + jc$
  - $j = i \cdot jc = jc \cdot i + 0$
  - $\rightarrow j = ja \cdot i + jb$ (avec ja et jb invariants de boucle)
- `j` est **fonction affine** de `i`

Hiérarchie de variable d'induction dérivée

- `k = j @ ko`
- Quelle est la relation mathématique entre j et k?

Hiérarchie de variable d'induction dérivée

- `k = j @ ko`
- Quelle est la relation mathématique entre j et k?
- Même affaire : $k = ka \cdot j + kb$
- $\rightarrow$ `k` est **fonction affine** de `j`

Et entre i et k ?

- $j = jb \cdot i + jb$, on déroule...

Hiérarchie de variable d'induction dérivée

- `k = j @ ko`
- Quelle est la relation mathématique entre j et k?
- Même affaire : $k = ka \cdot j + kb$
- $\rightarrow$ `k` est **fonction affine** de `j`

Et entre i et k ?

- $j = jb \cdot i + jb$, on déroule...

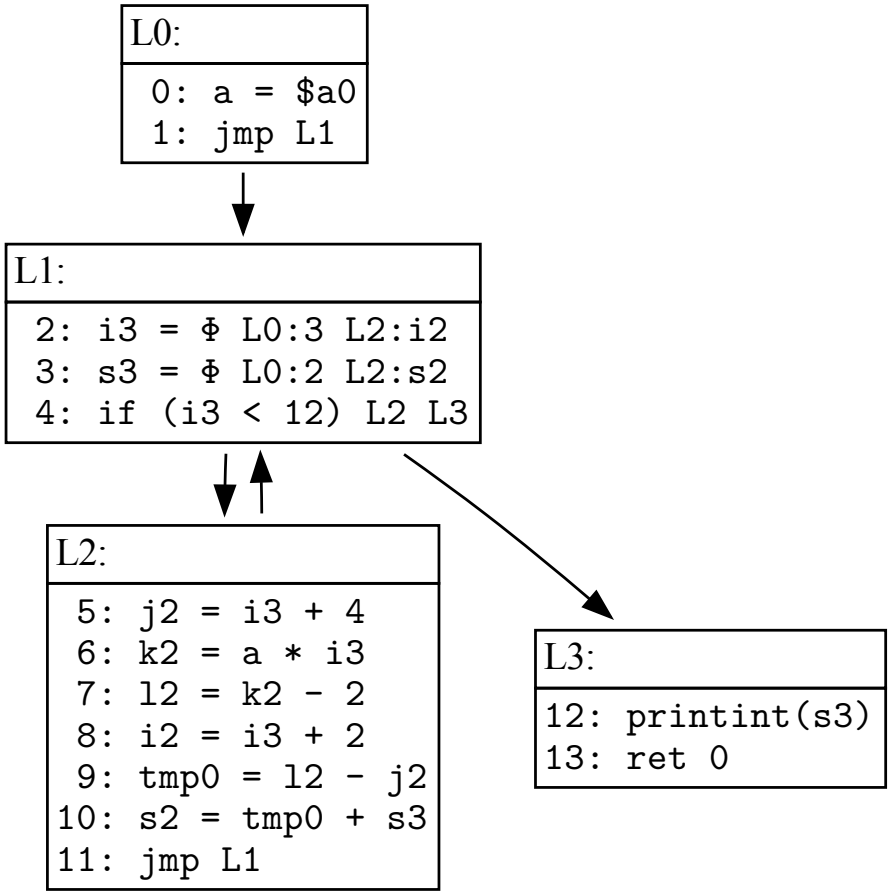
$$\begin{aligned} k &= ka \cdot j + kb \\ &= ka \cdot (ja \cdot i + jb) + kb \\ &= (ka \cdot ja) \cdot i + (ka \cdot jb + kb) \end{aligned}$$

- $\rightarrow$ `k` est aussi **fonction affine** de `i`

Hiérarchies

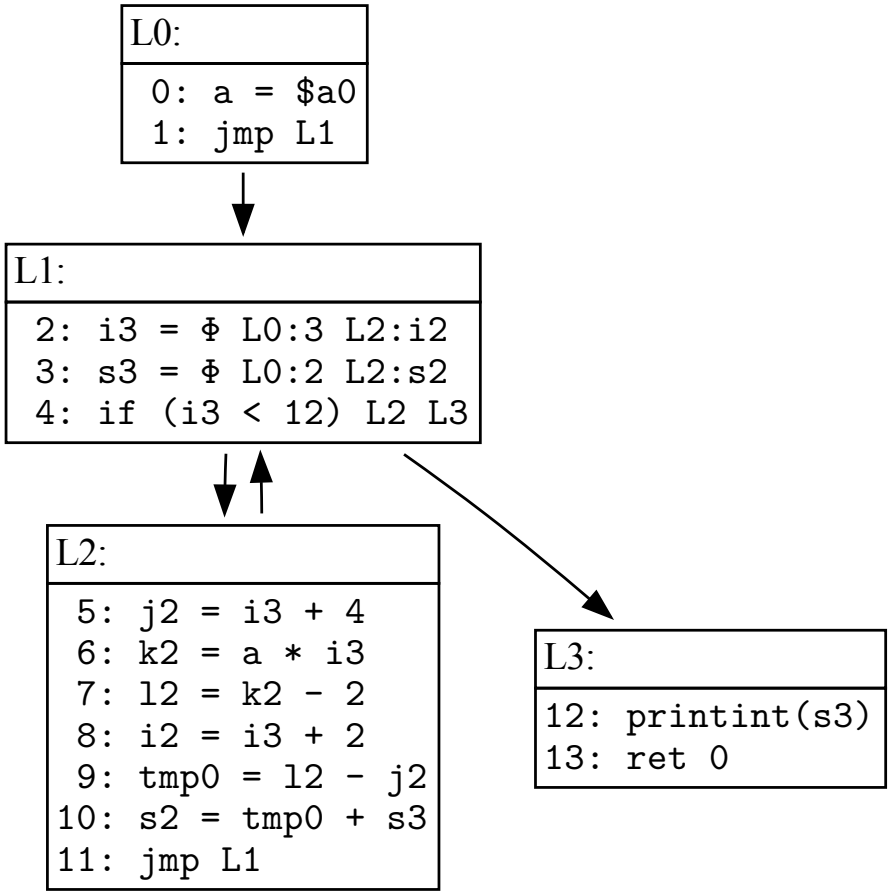
- Pour chaque variable d'induction basique `i`
- $\rightarrow$ Une **hiérarchie** de variables d'induction dérivées

Exercice : trouver les variables d'induction

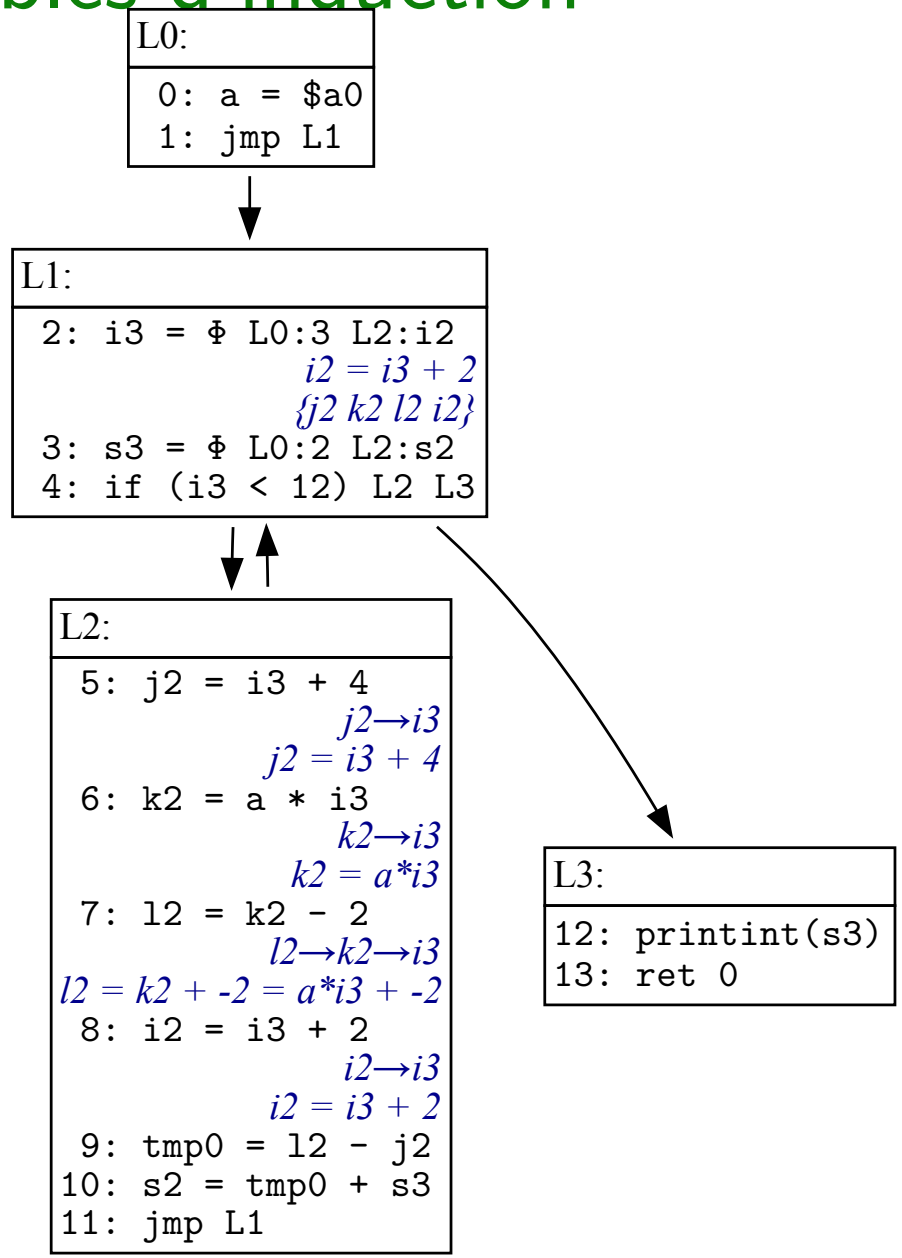


induction.c: foo()

Exercice : trouver les variables d'induction



induction.c: foo()



induction.c: foo()

Réduction des variables d'induction

Réduction des variables d'induction

Objectif

- Transformer `k = j @ ko` en `k = k + kc`
  Où `kc` est aussi un invariant de boucle

Gains

Réduction des variables d'induction

Objectif

- Transformer `k = j @ ko` en `k = k + kc`
  Où `kc` est aussi un invariant de boucle

Gains

- *Strength reduction*
  - Une multiplication en moins quand `@` est `*`
- Ça dissocie `k`, `j` (et `i`)
  - Meilleur parallélisme processeur
  - Voire éliminer `j` (et `i`) s'il deviennent inutile

Pertes

- Calculer et maintenir
  - une valeur initiale de `k`
  - un incrément `kc`

Esquisse de la stratégie

On a

- i_0 valeur initiale de i, connue en entrée de boucle
- $i_n = i_{n-1} + ic$ et $k_n = i_n \cdot ka + kb$ pour le n^{e} tour de boucle

On déroule k_n

Esquisse de la stratégie

On a

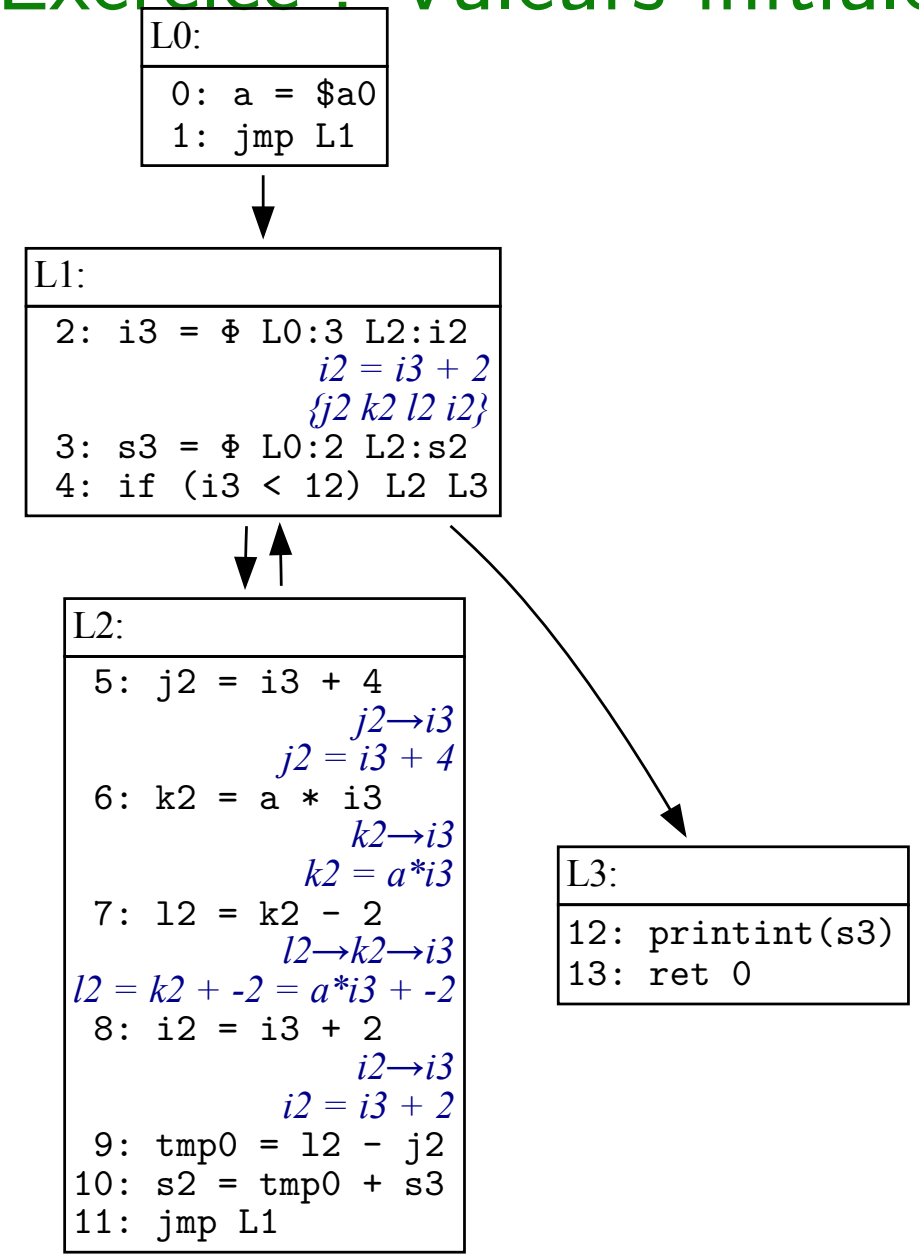
- i_0 valeur initiale de i, connue en entrée de boucle
- $i_n = i_{n-1} + ic$ et $k_n = i_n \cdot ka + kb$ pour le n^{e} tour de boucle

On déroule k_n

$$\begin{aligned} k_n &= i_n \cdot ka + kb \\ &= (i_{n-1} + ic) \cdot ka + kb \\ &= i_{n-1} \cdot ic + ka \cdot ic + kb \\ &= (i_{n-1} \cdot ka + kb) + (ka \cdot ic) \\ &= k_{n-1} + (ka \cdot ic) = k_{n-1} + kc \end{aligned}$$

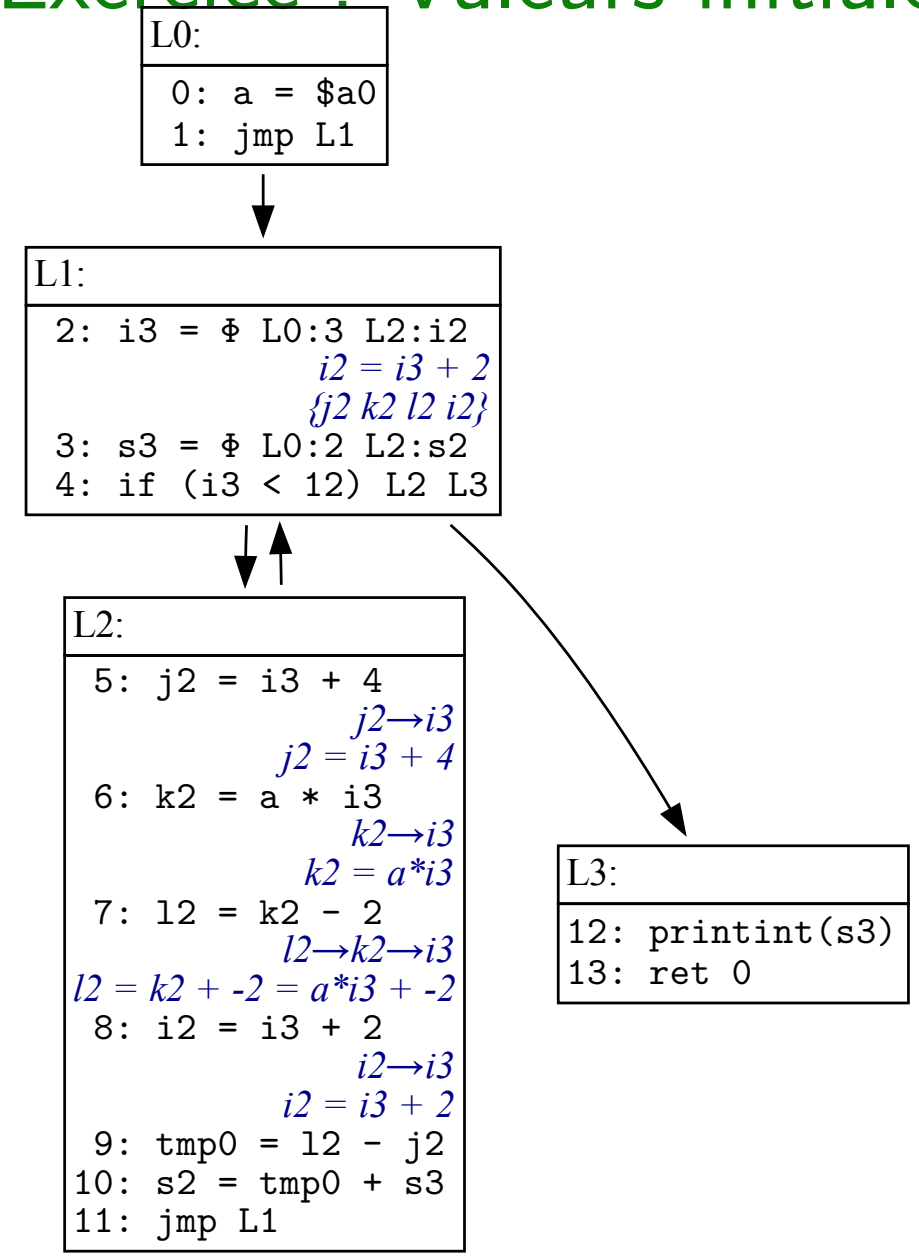
- Avec $k_0 = i_0 \cdot ka + kb$ et $kc = ka \cdot ic$
- k_0 et kc sont aussi des invariants de boucle
  donc calculables dans la pré-entête

Exercice : Valeurs initiales et incréments

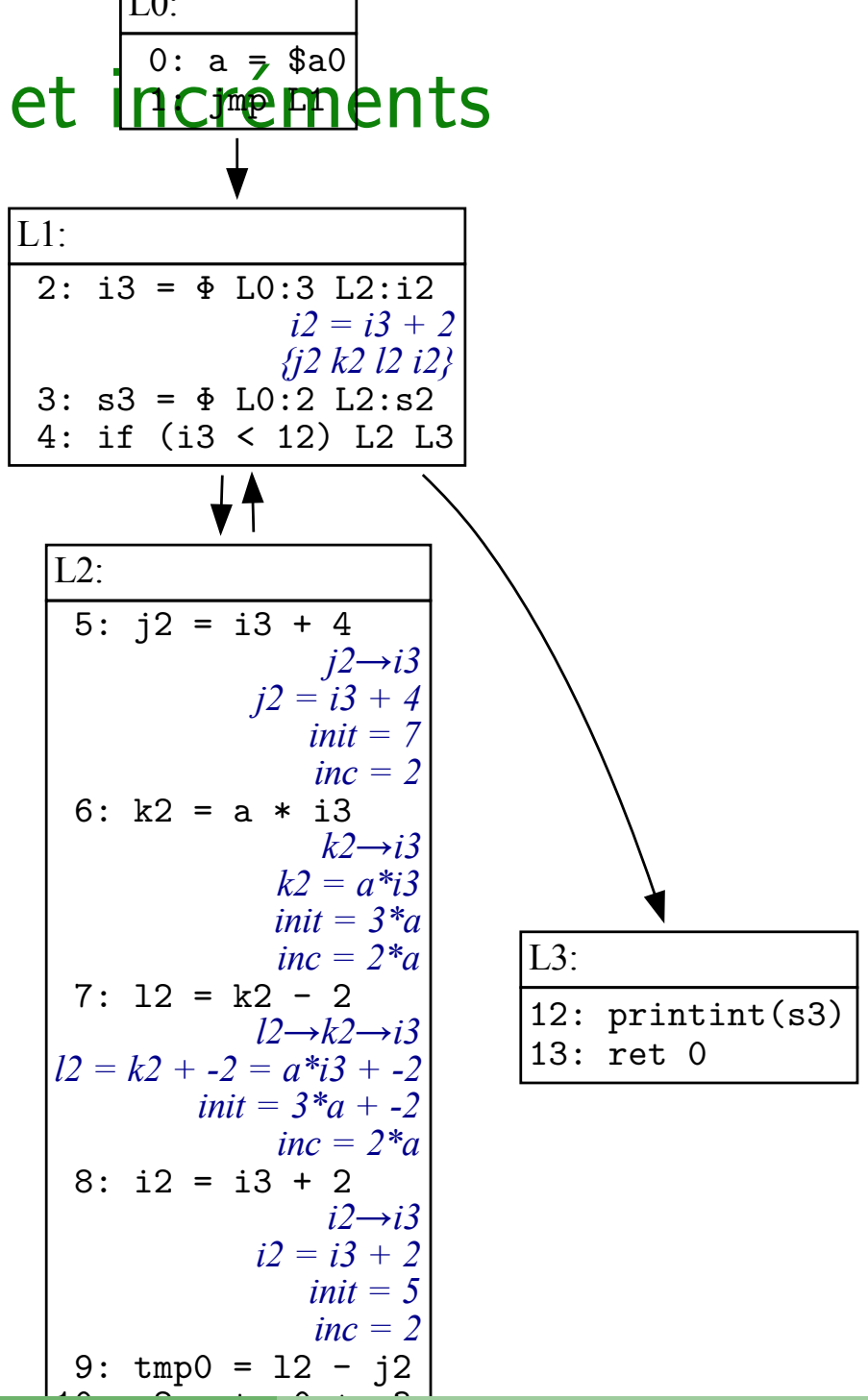


induction.c: foo()

Exercice : Valeurs initiales et incréments



induction.c: foo()



Mise en œuvre ?

On a

- Basique : `i = i + ic`, `i0` valeur initiale
- Dérivée : `j = i @ jo` $\rightarrow j = i \cdot ja + jb$

Valeur initiale et incrément

- Initial : $j0 = i0 \cdot ja + jb$ (ou `j0 = i0 @ jo`)
- Incrément : $jc = ja \cdot ic$
- `j0` et `jc` sont des invariants de boucle
- $\rightarrow$ Calculables dans la pré-entête

Magie

- Mettre `j = j0` dans la pré-entête
- Mettre `j = j + jc` dans la boucle
- Enlever `j = i @ jo` de la boucle
- ...
- Profit

Mise en œuvre ?

On a

- Basique : `i = i + ic`, `i0` valeur initiale
- Dérivée : `j = i @ jo` $\rightarrow j = i \cdot ja + jb$

Valeur initiale et incrément

- Initial : $j0 = i0 \cdot ja + jb$ (ou `j0 = i0 @ jo`)
- Incrément : $jc = ja \cdot ic$
- `j0` et `jc` sont des invariants de boucle
- $\rightarrow$ Calculables dans la pré-entête

Magie

- Mettre `j = j0` dans la pré-entête
- Mettre `j = j + jc` dans la boucle
- Enlever `j = i @ jo` de la boucle
- ...
- Profit ?

Problèmes

Les maths

- Ça va a peu près

L'implémentation

- Vraiment pas si facile
- Beaucoup de détails
  ... Beaucoup de diables

→ `InductionVariableElimination.java`

Diable : SSA

- 1: Φ pour les variables d'induction basiques
  - Donc deux registres de base au lieu d'un ? Ça complique...
- 2: Comment insérer `j = j + jc` ? C'est pas SSA

Diable : SSA

- 1: Φ pour les variables d'induction basiques
  - Donc deux registres de base au lieu d'un ? Ça complique...
- 2: Comment insérer `j = j + jc` ? C'est pas SSA

Approche 1

- Créer (au moins) deux variables pour `j`
  - Avant et après incrément
- Insérer les Φ aux bons endroits
  - Respecter la dominance
  - Éviter les Φ inutiles

Approche 2

- Générer du code non-SSA
- Repasser un coup de transformation en SSA
- $\rightarrow$ Moins efficace ? Pas si sûr
- $\rightarrow$ Moins prise de tête ? Définitivement

Diable : Les invariants ne sont pas si simples

- On a `k = j @ op`
- On veut $k = ka \cdot i + kb$ et $k = k + kc$
- $\rightarrow$ facteur ka, décalage kb, valeur initiale $k0$, incrément kc ?

Nos défis

- On est en code 3 adresses
- Combinaison d'opérations et de variables intermédiaires
- Les invariants de boucles ne sont pas forcément des constantes

Structures de données dédiées symboliques

- Avec les soucis d'ingénierie qui vont avec
- Et pouvoir générer de l'IR à partir d'eux
  - Générer le code qui initialise $k0$, kc
- $\rightarrow$ voir `Sum.java` et `Product.java`

Diable : Où mettre les « nouvelles » instructions ?

- Avant ? Après l'incrément ? de `k` ? de `j` ? de `i` ? Ailleurs ?
- Les variables visibles/utilisables changent d'un endroit à l'autre
- En en SSA elles changent de noms !

Dissocier

Pour $j = ja \cdot i + jb$, on a deux choses

- La valeur de $ja \cdot i + jb$
  - $\rightarrow$ « change » au moment de l'incrément de `i`
- La valeur de `j`
  - $\rightarrow$ « change » au moment de l'affectation de `j`

Solution

- Créer un accumulateur `js` **synchronisé** avec les tours de boucles
- Capturer `j` au moment de son affectation

Mise en œuvre qui fonctionne

Rappels et notations

Variable d'induction basique

- `i2 = i1 + ic`
- `i0` valeur initiale
- `i1` valeur pré-incrément
- `i2` valeur post-incrément
- `ic` valeur de l'incrément

Variable d'induction dérivée

- `j = i1 @ jo` $\rightarrow j = i \cdot ja + jb$
- Valeur initiale : $j0 = i0 \cdot ja + jb$ (ou `j0 = i0 @ jo`)
- Valeur de l'incrément : $jc = ja \cdot ic$

Mise en œuvre qui fonctionne

Valeur initiale (et accumulateur)

- Introduire `js` accumulateur de `j`
- Initialiser dans la pré-entête `js = j0`
- Initialiser `jc` dans la pré-entête (si nécessaire)

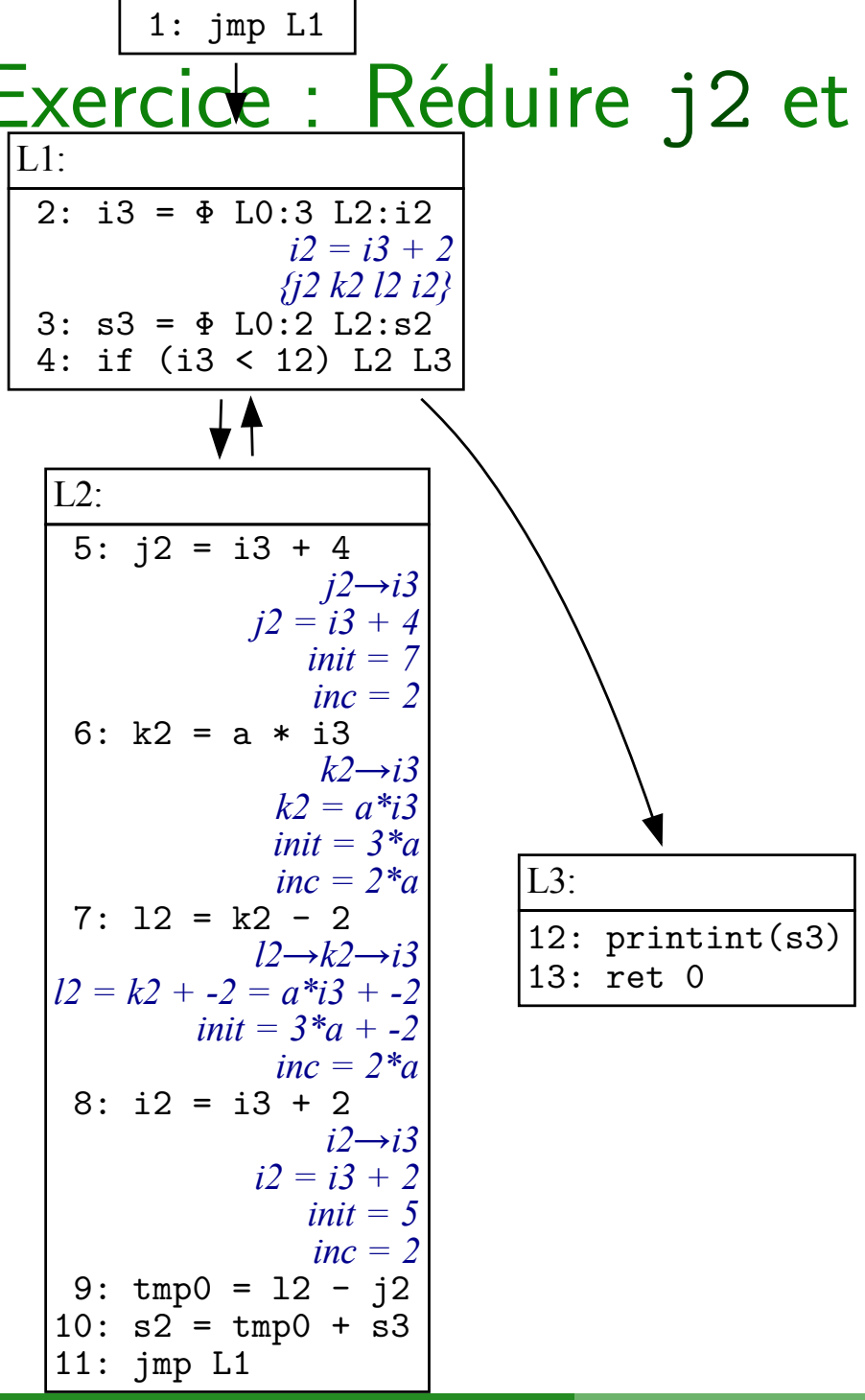
Définition de `j` **au bon moment**

- Remplacer `j = i @ jo`
- Par `j = js` : capture la valeur de `j`

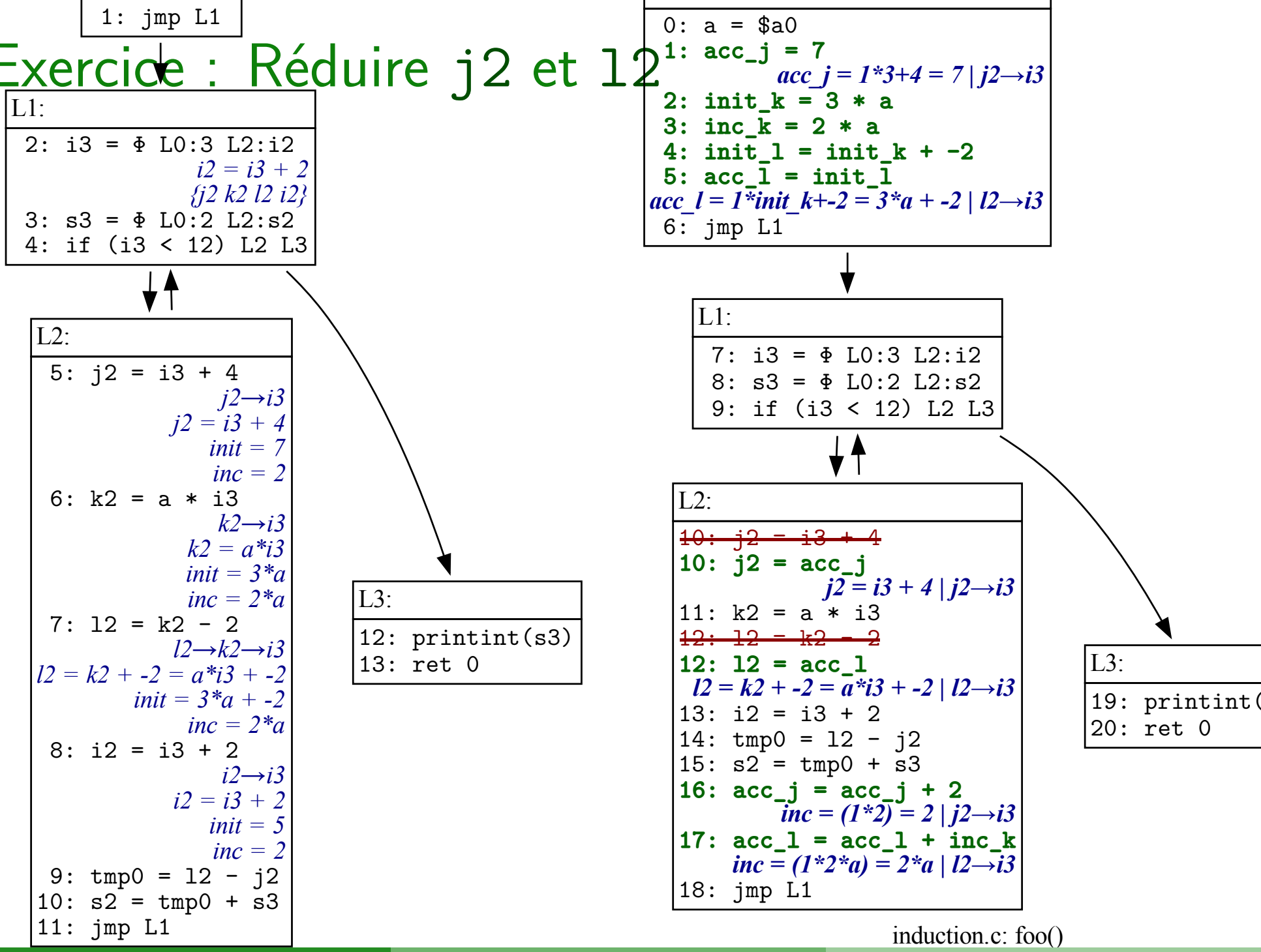
Incrément **au bon moment**

- Insérer `js = js + jc` à la fin de la queue de boucle

Exercice : Réduire j2 et l2



Exercice : Réduire j2 et l2



induction.c: foo()

Diable : Quelles variables d'induction réduire ?

Power réduction

- Éliminer une multiplication c'est bien, non ?
- Est-ce *vraiment* toujours bien ?

Dead code elimination

- Faire disparaître des variables d'inductions devenues inutiles
  - Car on a réduit celles qui en **dépendent**
- Une variable d'induction est-elle utile par ailleurs ?
- $\rightarrow$ Si elle sert à autre chose que calculer d'autres variables d'inductions

Diable : Quelles variables d'induction réduire ?

- On y perd parfois plus qu'on y gagne

Diable : Quelles variables d'induction réduire ?

- On y perd parfois plus qu'on y gagne

Gain

- `j = j @ jo` devient `j = js`
- Qui n'est pas forcément dans le chemin critique

Perte

- Un registre `js` (voire `jc`) à maintenir vivant
- Ajout de `js = js + jc` dans le chemin critique de la boucle
- Ajout de l'initialisation de `js` et `jc` dans la pré-entête

Diable : Variables post-incrément

- Que faire de `j = i2 @ io` ?
- Où `i2` est la variable d'induction de base post-incrément ?
- i.e. `i2 = i1 + ic` pour l'incrément de `i`

Diable : Variables post-incrément

- Que faire de `j = i2 @ io` ?
- Où `i2` est la variable d'induction de base post-incrément ?
- i.e. `i2 = i1 + ic` pour l'incrément de `i`

Traiter comme un cas à part complexe

- Variable d'induction basique
  - $\rightarrow$ Concerne en fait deux registres
- Variable d'induction dérivée
  - $\rightarrow$ Se rappeler lequel des deux est utilisé
- Incrémenter `js` quand il faut
  - Début ou fin de boucle

Ou utiliser une astuce

- Considérer que `i2` est **aussi** une variable d'induction **dérivée**
- Basée sur la variable d'induction de **base** `i1`
- `j` est donc une variable dérivée seconde

Diable : Séquences d'incréments

Incrément des accumulateurs

- Ont tous lieu aux même endroit

Risque

Diable : Séquences d'incréments

Incrément des accumulateurs

- Ont tous lieu aux même endroit

Risque

- Ajout de pleins d'incréments
  - $\rightarrow$ Un pour chaque variable dérivée réduite
- Tout en plein dans la boucle
  - $\rightarrow$ Dans le chemin critique

Diable : Se remplir la tête

Remonter un maximum d'invariants

- Les invariant originaux (LICM)
- Les invariants des réductions
- $\rightarrow$ Dans la **pré-entête**, qui ne fait pas partie de la boucle
- $\rightarrow$ Un seul calcul quelque soit le nombre de tours de la boucle

Risque

Diable : Se remplir la tête

Remonter un maximum d'invariants

- Les invariant originaux (LICM)
- Les invariants des réductions
- $\rightarrow$ Dans la **pré-entête**, qui ne fait pas partie de la boucle
- $\rightarrow$ Un seul calcul quelque soit le nombre de tours de la boucle

Risque

Quand la condition de la boucle est initialement fausse

- La boucle n'est pas exécutée
- On fait zéro tour
- On ne passe pas Go
- On ne touche pas 200 $
- Mais, on a calculé tous ces trucs dans la pré-entête pour rien
- Solution ? $\rightarrow$ Rotation de boucle ? (plus loin)

Élimination des variables d'induction

Gestion de boucle

- Youpi, `j` s'est fait réduire

```
js = ...;
jc = ...;
for(int i=0; i<l; i++) { // seules occurences de i
  ...;
  j = js;
  ...;
  js = js + jc; // on sait que js = ja * i + jb
}
```

- Maintenant, `i` ne sert ici qu'a la gestion de la boucle
- Peut-on s'en débarrasser ?

Élimination des variables d'induction

On a

- $j_n = aj \cdot i_n + bj$
- Peut on améliorer $i_n < l$? On déroule...

Élimination des variables d'induction

On a

- $j_n = aj \cdot i_n + bj$
- Peut on améliorer $i_n < l$? On déroule...

$$
\begin{aligned}
i_n < l &\Leftrightarrow aj \cdot i_n < aj \cdot l \quad \text{et } aj > 0 \\
&\Leftrightarrow aj \cdot i_n + bj < aj \cdot l + bj \quad \text{et } aj > 0 \\
&\Leftrightarrow j_n < aj * l + bj \quad \text{et } aj > 0
\end{aligned}
$$

Signe de aj

- Si $aj < 0$, on peut inverser la comparaison
- Si on connaît pas le signe de aj on ne peut rien faire :(

Mise en œuvre

Déterminer le signe de aj

- Sous problème en soit
- Si aj est une constante immédiate, ça va
- Si aj est un `unsigned`, ça va aussi :p
- Si aj dépend d'invariants de boucle c'est plus compliqué
  - $\rightarrow$ Analyse de signes (voir INF889A)

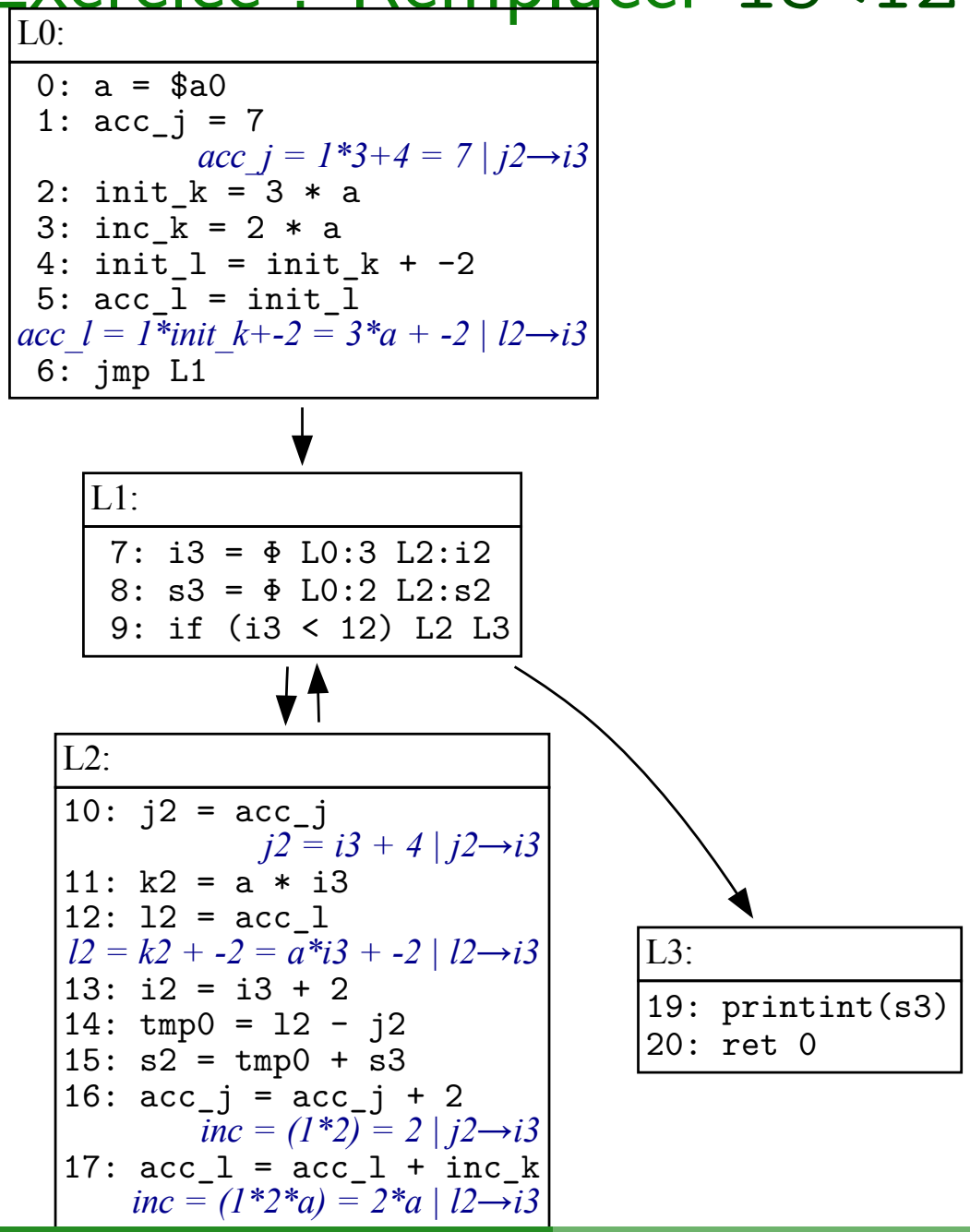
Calculer la nouvelle limite

- $jl = aj \cdot l + bj$ dans la pré-entête

Remplacer et nettoyer

- `i < l` devient `js < jl` dans le `if`
- `i` est ensuite éliminé lors d'une passe suivante

Exercice : Remplacer i3<12 avec acc_j<???



Exercice : Remplacer i3<12 avec acc_j<???

L0:

```
0: a = $a0
1: acc_j = 7
```

*acc_j = 1*3+4 = 7 | j2→i3*

```
2: init_k = 3 * a
3: inc_k = 2 * a
4: init_l = init_k + -2
5: acc_l = init_l
```

*acc_l = 1*init_k+-2 = 3*a + -2 | l2→i3*

```
6: jmp L1
```

L1:

```
7: i3 = Φ L0:3 L2:i2
8: s3 = Φ L0:2 L2:s2
9: if (i3 < 12) L2 L3
```

L2:

```
10: j2 = acc_j
```

j2 = i3 + 4 | j2→i3

```
11: k2 = a * i3
12: l2 = acc_l
```

*l2 = k2 + -2 = a*i3 + -2 | l2→i3*

```
13: i2 = i3 + 2
14: tmp0 = l2 - j2
15: s2 = tmp0 + s3
16: acc_j = acc_j + 2
```

*inc = (1*2) = 2 | j2→i3*

```
17: acc_l = acc_l + inc_k
```

*inc = (1*2*a) = 2*a | l2→i3*

L3:

```
19: printint(s3)
20: ret 0
```

L0:

```
0: a = $a0
1: acc_j = 7
2: init_k = 3 * a
3: inc_k = 2 * a
4: init_l = init_k + -2
5: acc_l = init_l
6: jmp L1
```

L1:

```
7: i3 = Φ L0:3 L2:i2
8: s3 = Φ L0:2 L2:s2
```

~~9: if (i3 < 12) L2 L3~~

9: if (acc_j < 16) L2 L3

limit:1*12+4 = 16 | j2→i3

L2:

```
10: j2 = acc_j
11: k2 = a * i3
12: l2 = acc_l
13: i2 = i3 + 2
14: tmp0 = l2 - j2
15: s2 = tmp0 + s3
16: acc_j = acc_j + 2
17: acc_l = acc_l + inc_k
18: jmp L1
```

L3:

```
19: printint(s3
20: ret 0
```

Nettoyage : SSA

L0:

```
0: a = $a0
1: acc_j = 7
2: init_k = 3 * a
3: inc_k = 2 * a
4: init_l = init_k + -2
5: acc_l = init_l
6: jmp L1
```

L1:

```
7: acc_l2 = Φ L0:acc_l L2:acc_l3
8: acc_j2 = Φ L0:acc_j L2:acc_j3
9: i3 = Φ L0:3 L2:i2
10: s3 = Φ L0:2 L2:s2
```

~~11: if (acc_j < 16) L2 L3~~

11: if (acc_j2 < 16) L2 L3

L2:

~~12: j2 = acc_j~~

12: j2 = acc_j2

13: k2 = a * i3

~~14: l2 = acc_l~~

14: l2 = acc_l2

15: i2 = i3 + 2

16: tmp0 = l2 - j2

17: s2 = tmp0 + s3

~~18: acc_j = acc_j + 2~~

18: acc_j3 = acc_j2 + 2

~~19: acc_l = acc_l + inc_k~~

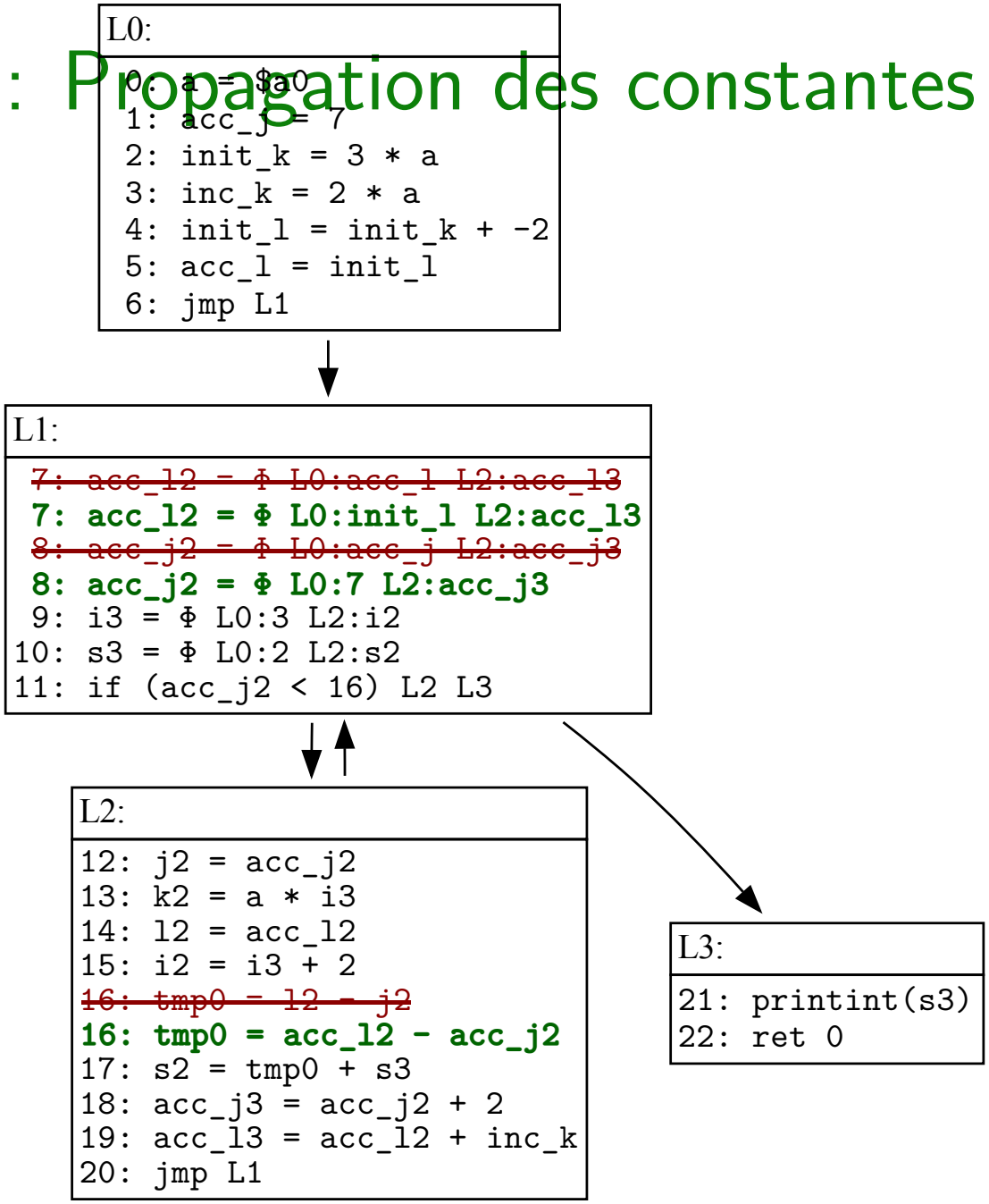
19: acc_l3 = acc_l2 + inc_k

20: jmp L1

L3:

```
21: printint(s3)
22: ret 0
```

Nettoyage : Propagation des constantes et copies



induction.c: foo()

Nettoyage : Élimination (agressive) du code mort

L0:

```
0: a = $a0
1: acc_j = 7
2: init_k = 3 * a
3: inc_k = 2 * a
4: init_l = init_k + -2
5: acc_l = init_l
6: jmp L1
```

L1:

```
 7: acc_l2 = Φ L0:init_l L2:acc_l3
 8: acc_j2 = Φ L0:7 L2:acc_j3
 9: i3 = Φ L0:3 L2:i2
10: s3 = Φ L0:2 L2:s2
11: if (acc_j2 < 16) L2 L3
```

L2:

```
12: j2 = acc_j2
13: k2 = a * i3
14: l2 = acc_l2
15: i2 = i3 + 2
16: tmp0 = acc_l2 - acc_j2
17: s2 = tmp0 + s3
18: acc_j3 = acc_j2 + 2
19: acc_l3 = acc_l2 + inc_k
20: jmp L1
```

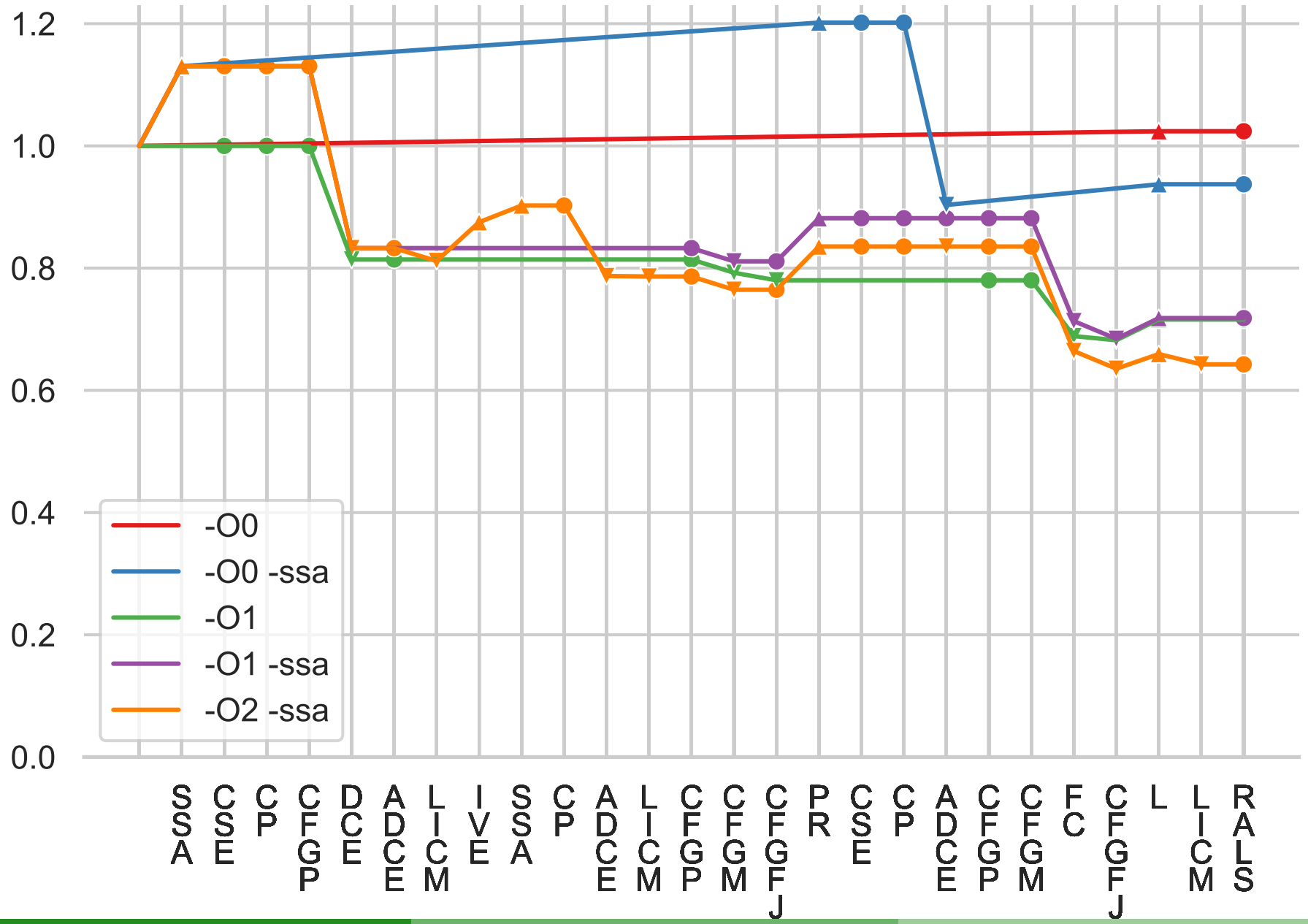
L3:

```
21: printint(s3)
22: ret 0
```

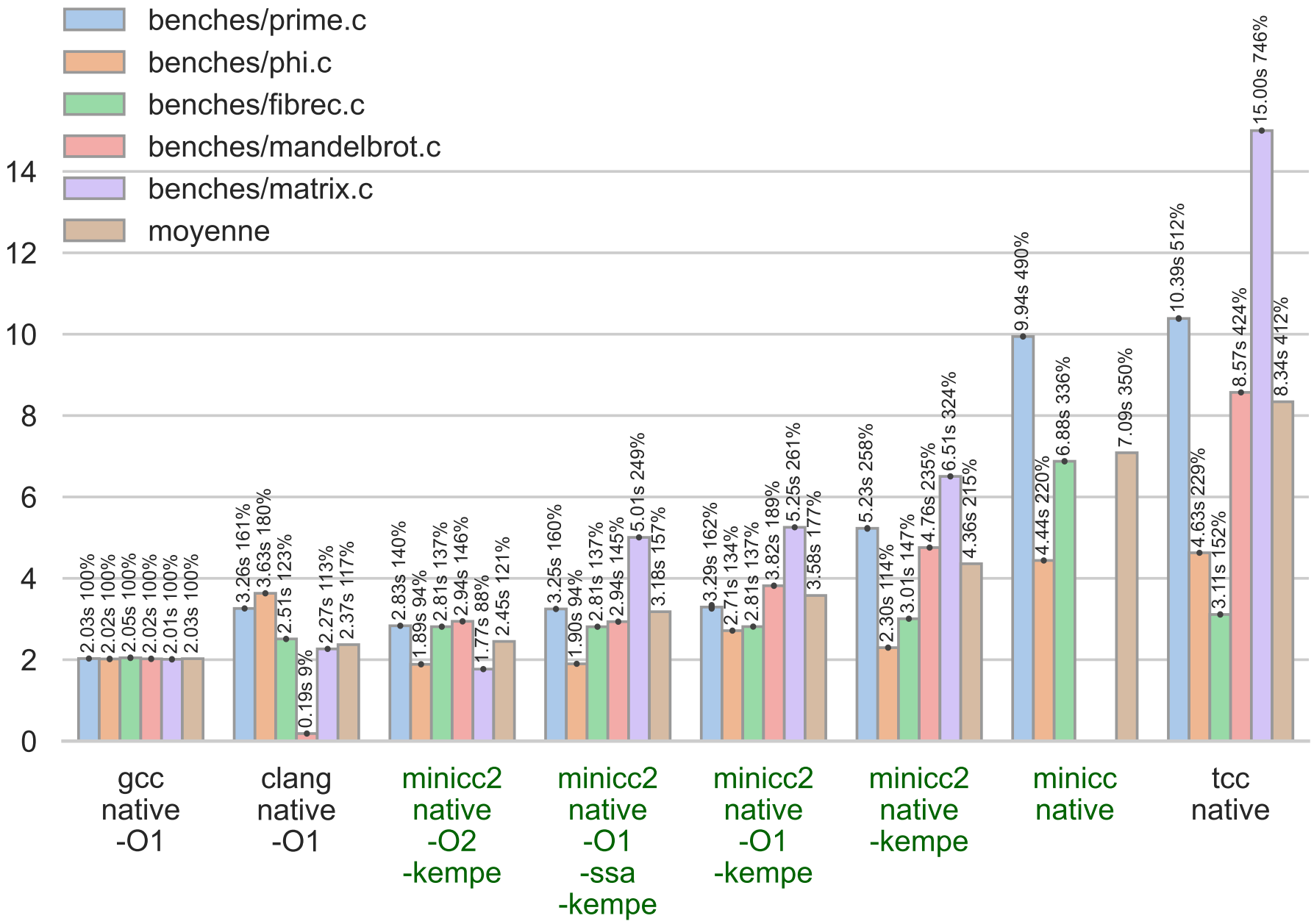
induction.c: foo()

MiniCC2 -O2

Estimation de coût



Métal 🤘


benches/prime.c
benches/phi.c
benches/fibrec.c
benches/mandelbrot.c
benches/matrix.c
moyenne
0
2
4
6
8
10
12
14
2.03s 100%
2.02s 100%
2.05s 100%
2.02s 100%
2.01s 100%
2.03s 100%
3.26s 161%
3.63s 180%
2.51s 123%
0.19s 9%
2.27s 113%
2.37s 117%
2.83s 140%
1.89s 94%
2.81s 137%
2.94s 146%
1.77s 88%
2.45s 121%
3.25s 160%
1.90s 94%
2.81s 137%
2.94s 145%
5.01s 249%
3.18s 157%
3.29s 162%
2.71s 134%
2.81s 137%
3.82s 189%
5.25s 261%
3.58s 177%
5.23s 258%
2.30s 114%
3.01s 147%
4.76s 235%
6.51s 324%
4.36s 215%
9.94s 490%
4.44s 220%
6.88s 336%
7.09s 350%
10.39s 512%
4.63s 229%
3.11s 152%
8.57s 424%
15.00s 746%
8.34s 412%
gcc native -O1
clang native -O1
minicc2 native -O2 -kempe
minicc2 native -O1 -ssa -kempe
minicc2 native -O1 -kempe
minicc2 native -kempe
minicc native
tcc native

La fin de MiniCC2 ?

- Ce n'est pas la **destination** qui compte
- Mais le **chemin** parcouru

- C'était amusant à implémenter !

Autres optimisations de boucles

Déroulage de boucle (*loop unrolling*)

```
for(i=0;i<n;i++)
  a[i] = b[i] + c[i];
```

On déroule (duplique) le corps de la boucle

Déroulage de boucle (*loop unrolling*)

```
for(i=0;i<n;i++)
  a[i] = b[i] + c[i];
```

On déroule (duplique) le corps de la boucle

```
for(i=0;i<n%4;i++)
  a[i] = b[i] + c[i];
for(i=0;i<n;i+=4) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

Gains ? Pertes ?

Gains ? Pertes ?

- Des branchements en moins
- Augmente la possibilité de parallélisme processeur
- Préalable à un meilleur ordonnancement des instructions (*software pipelining*)
- Préalable à des optimisations vectorielles
- Augmentation de la taille du programme

Parallélisation de boucle (*loop parallelisation*)

- Programme précédent
- On suppose `a`, `b` et `c` indépendants

On utilise une instruction SIMD

Single instruction multiple data

Parallélisation de boucle (*loop parallelisation*)

- Programme précédent
- On suppose a, b et c indépendants

On utilise une instruction SIMD

Single instruction multiple data

```
//a b et c sont indépendants
for(i=0;i<n%4;i++)
  a[i] = b[i] + c[i];
for(i=0;i<n;i+=4)
  __SIMD4add(a+i, b+i, c+i)
```

Pelage de boucle ? (*loop peeling*)

```
for(i=0;i<n;i++) {
  if(i==0) b[i] = a[i];
  else b[i] = a[i]+b[i-1];
}
```

On déroule seulement la (les) première(s) itérations

Pelage de boucle ? (*loop peeling*)

```
for(i=0;i<n;i++) {
  if(i==0) b[i] = a[i];
  else b[i] = a[i]+b[i-1];
}
```

On déroule seulement la (les) première(s) itérations

```
b[i] = a[i];
for(i=1;i<n;i++) {
  b[i] = a[i]+b[i-1];
}
```

Gains ? Pertes ?

Fusion de boucles (*loop fusion*)

```
for(i=0;i<n;i++) b[i] = a[i];
for(j=0;j<n;j++) c[j] = b[j];
```

On combine deux boucles équivalentes ensemble

Fusion de boucles (*loop fusion*)

```
for(i=0;i<n;i++) b[i] = a[i];
for(j=0;j<n;j++) c[j] = b[j];
```

On combine deux boucles équivalentes ensemble

```
for(i=0;i<n;i++) { b[i] = a[i]; c[i] = b[i]; }
```

Ce qui permet ensuite

```
for(i=0;i<n;i++) { c[i] = b[i] = a[i]; }
```

Gains ? Pertes ?

Rotation de boucle (*loop rotation*)

```
while(b) {
  ...
}
```

On met la condition à la fin

Rotation de boucle (*loop rotation*)

```
while(b) {
  ...
}
```

On met la condition à la fin

```
if(b) {
  do {
    ...;
  } while (b);
}
```

Gain ? Pertes ?

Effondrement de boucle ? (*loop collapse*)

```
for(i=0;i<n;i++) {}
```

On supprime les boules inutiles

- ADCE s'en occupe en général

Échange de boucles (*loop interchange*)

```
for(i=0;i<10;i++)
  for(j=0;j<10;j++)
    b[j] += a[j][i] * x[j];
```

On inverse l'imbrication de deux boucles

Échange de boucles (*loop interchange*)

```
for(i=0;i<10;i++)
  for(j=0;j<10;j++)
    b[j] += a[j][i] * x[j];
```

On inverse l'imbrication de deux boucles

```
for(j=0;j<n;j++)
  for(i=0;i<n;i++)
    b[j] += a[j][i] * x[j];
```

Gains ? Pertes ?

Pavage de boucles (*loop tiling*)

Briser les boucles en petit morceaux

- Créations de boucles imbriqués supplémentaires
- Mais augmenter la localité spatiale
- $\rightarrow$ Pour tenir mieux dans des caches mémoire

Compilation polyédrique (*polyhedral compilation*)

- Ancourt&Irigoin,1991
- Vient des supercalculateurs et du parallélisme
- Programmation linéaire en nombres entiers (*integer linear programming*, ILP)
- $\rightarrow$ Donne du **travail** aux algébriens linéaires

En très simpliste

- Identifier les relations affines entre éléments des boucles
  - Dans l'espace : à travers le code
  - Dans le temps : à travers les tours de boucles
- Modéliser et optimiser tout ça (ILP)
- Trouver des boucles équivalentes performantes et parallélisables
- Toujours plus ou moins expérimental, mais on y croit fort !
  - gcc (Graphite)
  - llvm (polly)

Plus d'analyses et de transformation

- Déterminer la légalité de la transformation
  - $\rightarrow$ Preuves
- Déterminer la pertinence de la transformation
  - $\rightarrow$ Estimations
- Faire la transformation sans tout briser
  - $\rightarrow$ Ingénierie

La prochaine fois

- Optimisations inter-procédurales
- Analyses de la mémoire
- Langages à objets