Cours de compilation

Chapitre 9. Ordonnancement

Master informatique fondamentale 1 – ENS-Lyon

Christophe Alias
Christophe.Alias@ens-lyon.fr
http://perso.ens-lyon.fr/christophe.alias

Introduction

- La sélection d'instruction produit un nouveau DAG dont les noeuds sont des instructions de la machine cible.
- Il faut maintenant trouver un ordre d'évaluation pour ces instructions.

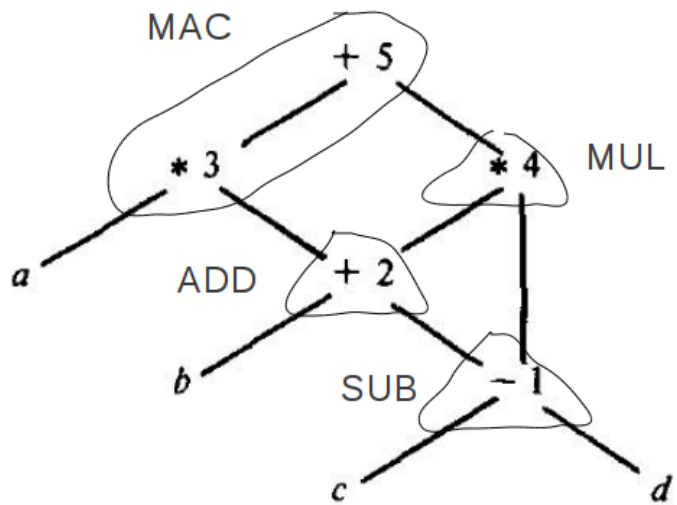
$$u_1 \leftarrow c - d$$

$$u_2 \leftarrow b + u_1$$

$$u_3 \leftarrow a * u_2$$

$$u_4 \leftarrow u_2 * u_1$$

$$u_5 \leftarrow u_3 + u_4$$



Ordonnancement

- Un ordonnancement est une fonction θ qui associe une date logique à chaque instruction.
- Pour être correct, l'ordonnancement doit respecter les dépendances de données: $i \rightarrow j \Rightarrow \theta(i) < \theta(j)$.
- Parmis les ordonnancements corrects, il faut trouver celui qui utilise au mieux les ressources de la machine cible (registres, prefetch, parallélisme, ...).

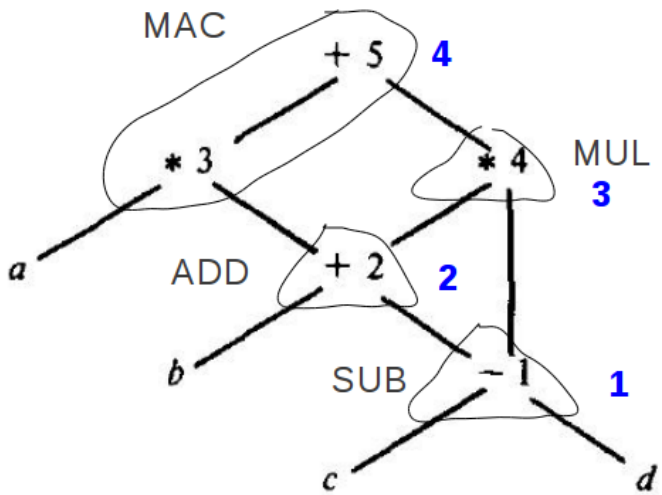
$$u_1 \leftarrow c - d$$

$$u_2 \leftarrow b + u_1$$

$$u_3 \leftarrow a * u_2$$

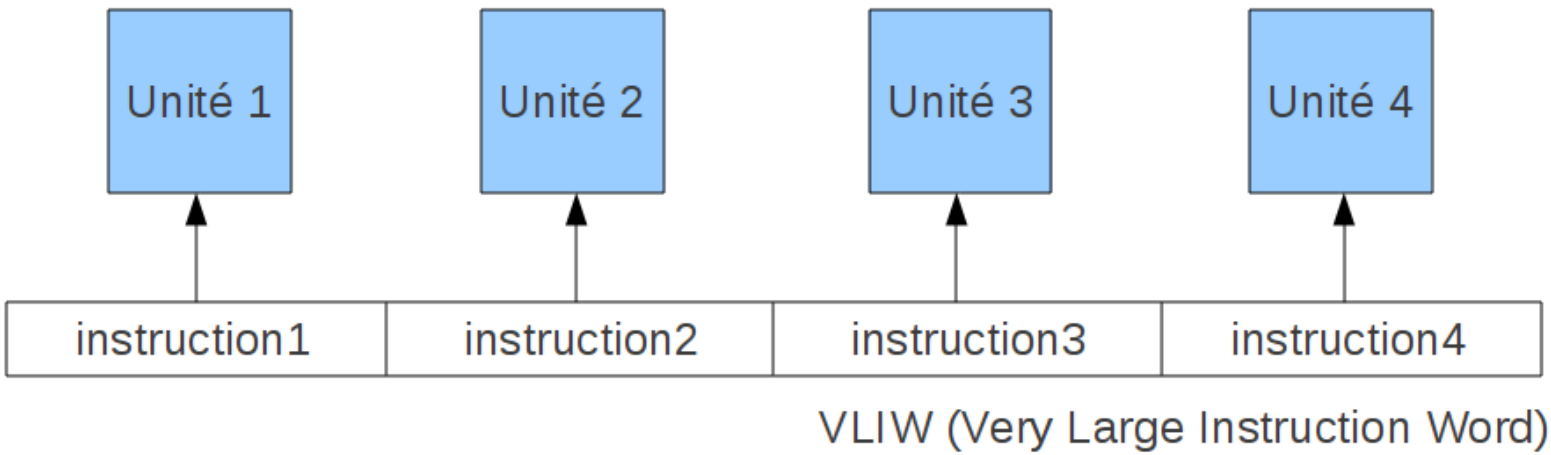
$$u_4 \leftarrow u_2 * u_1$$

$$u_5 \leftarrow u_3 + u_4$$



Exemple 1: Parallélisme d'instruction

- Certaines architectures comportent plusieurs unités fonctionnelles utilisables en parallèle.
- Une instruction est une concaténation de sous-instructions à exécuter en paralléle sur les différentes unités fonctionnelles.
- Quel ordonnancement?



Exemple 2: Prefetch

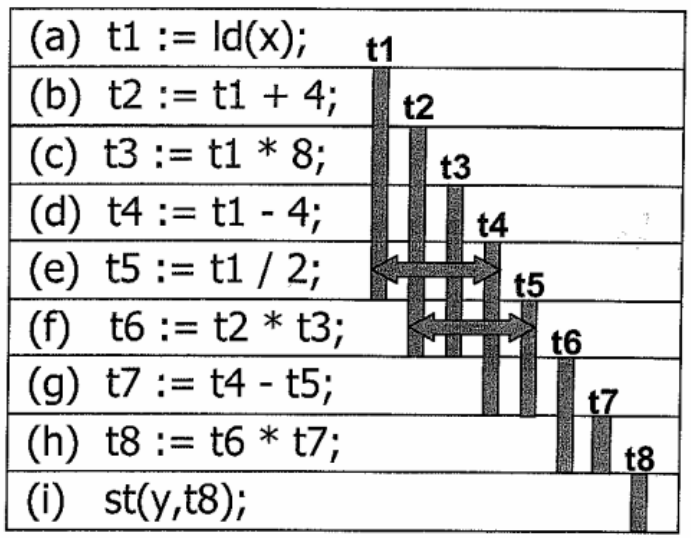
- Certaines architectures fournissent des instructions d'entrée/sortie (`load/store`) non-bloquantes.
- Il est alors possible d'exécuter des instructions entre un `load` et son utilisation, ce qui permet de cacher les latences d'accés mémoire.
- Quel ordonnancement?

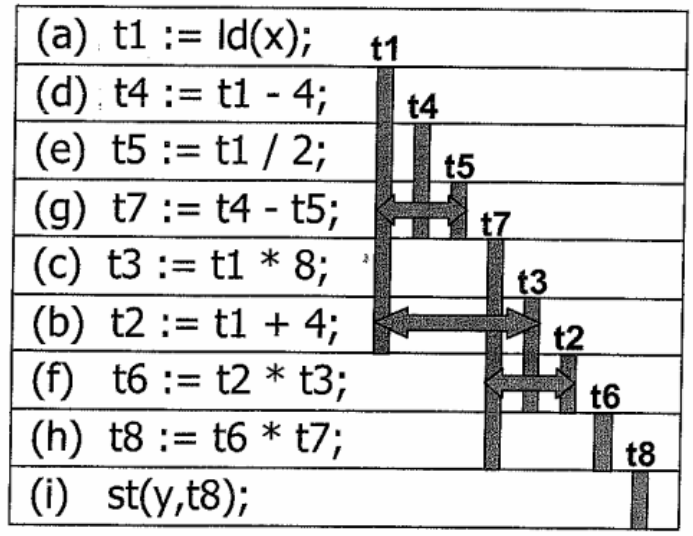
```
load r1,[ARP+8]
r2 = r1 + 1
r3 = r4 * r6
r6 = r3 – 2
```

```
load r1,[ARP+8]
r3 = r4 * r6
r6 = r3 – 2
r2 = r1 + 1
```

Exemple 3: Utilisation des registres

- La répartition des durées de vie des temporaires peut être modifiée pour réduire le besoin en registres (pression registre).
- Quel ordonnancement?





Quel critère d'ordonnancement?

Ces critères sont parfois incompatibles!
Exemple: prefetch & parallélisme *vs* besoin en registre.

Dans ce cours, on cherche un ordonnancement qui réduit le besoin en registres.

C'est un problème linéaire sur les arbres, mais NP-complet sur les DAGs [Sethi, 1975]...

Sur les arbres: Sethi-Ulmann

On décore chaque noeud de l'arbre avec le nombre de registres necéssaires à son calcul:

$$\begin{aligned} \rho(n) &= 1 \text{ si n est une feuille} \\ \rho(e_1\ op\ e_2) &= \begin{cases} \max\{\rho(e_1), \rho(e_2)\} & \text{si } \rho(e_1) \neq \rho(e_2) \\ \max\{\rho(e_1), \rho(e_2)\} + 1 & \text{si } \rho(e_1) = \rho(e_2) \end{cases} \end{aligned}$$

Une feuille necéssite un registre pour stocker sa valeur.

Sur un noeud intermédiaire, si une branche nécessite plus de registres que l'autre, on l'exécute en premier. Il reste alors assez de registres pour exécuter l'autre branche, et appliquer le calcul du noeud.

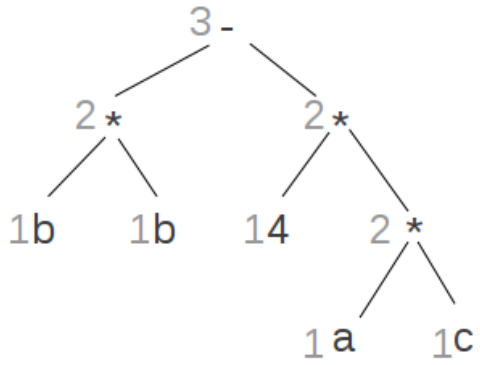
Si les deux branches nécessitent autant de registres, il faudra un registre supplémentaire pour stocker le résultat de la première branche.

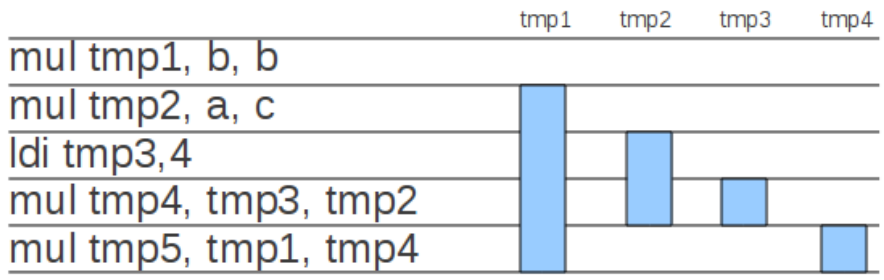
Sur les arbres: Sethi-Ulmann

On produit ensuite le code avec un parcours post-fixe de l'arbre, qui visite les noeuds les plus consommateurs en registres en premier:

```
function labelfs(node v)
(* generates an optimal evaluation for the subtree with root v *)
begin
if v is not a leaf
then if label(lson(v)) > label(rson(v))
       then labelfs(lson(v)); labelfs(rson(v))
       else labelfs(rson(v)); labelfs(lson(v))
       fi
fi
reg(v) ← new_reg(); print(v, reg(v));
if v is not a leaf then regfree(reg(lson(v))); regfree(reg(rson(v))) fi
end labelfs;
```

Exemple

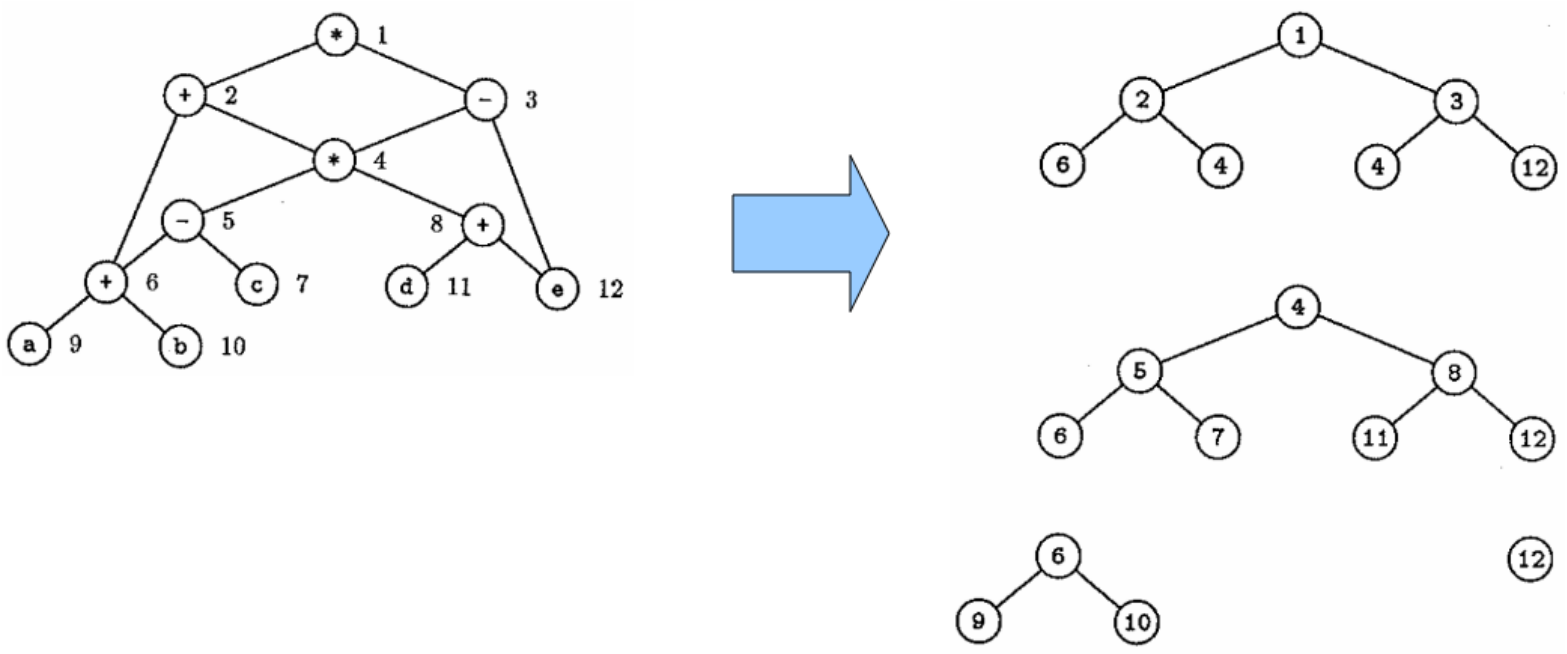

3 -
2 *
2 *
1b
1b
14
2 *
1 a
1c


tmp1
tmp2
tmp3
tmp4
mul tmp1, b, b
mul tmp2, a, c
ldi tmp3,4
mul tmp4, tmp3, tmp2
mul tmp5, tmp1, tmp4

Et sur les DAGs?

Heuristique naïve: on divise le DAGs en sous-arbres qui ne contiennent pas de noeuds partagés, sauf sur les feuilles

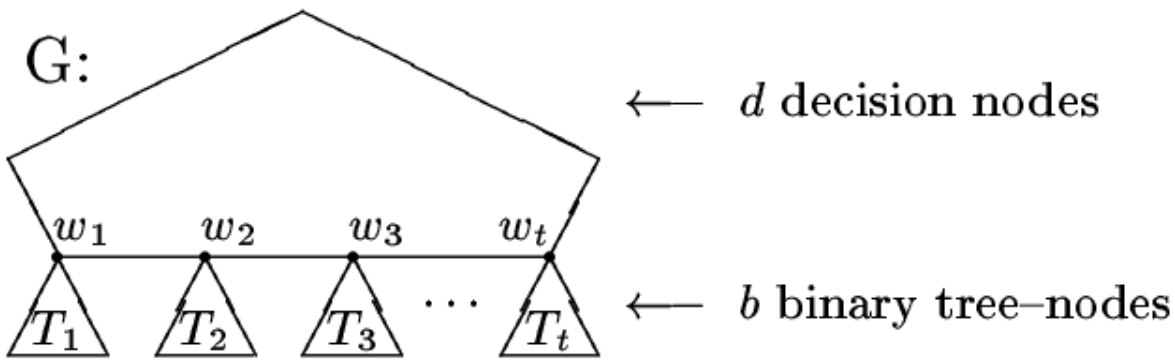
On applique ensuite Sethi-Ullman en suivant l'ordre des dépendances entre arbres.



Une heuristique randomisée

On partitionne le DAG en:

- tree-node: noeud d'un arbre "feuille" du DAG
- decision-node: les autres noeuds

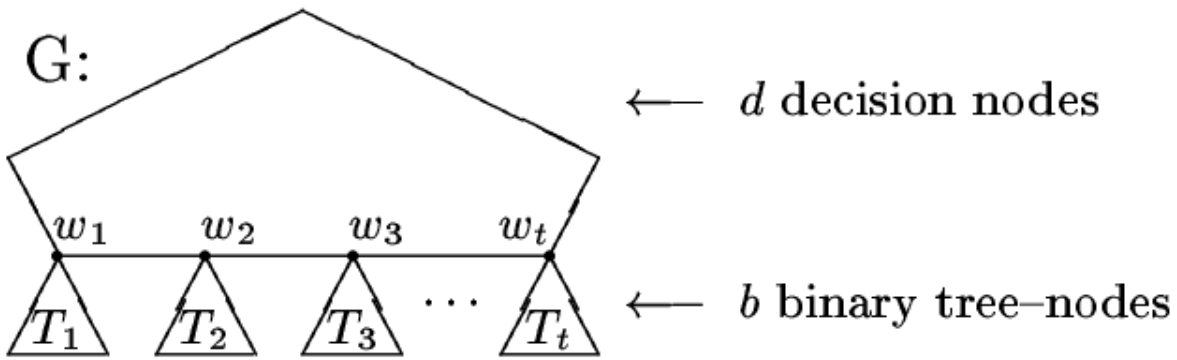


Une heuristique randomisée

On peut ordonnancer chaque tree-node avec Sethi-Ullman.

Pour les decision-nodes, on peut tester plusieurs possibilités et garder la meilleure.

Une possibilité peut s'encoder sous la forme d'un masque binaire 1100011 ou le bit i vaut 0 si on génére d'abord le fils gauche du ième décision-node et 1 sinon.



Ordonnancement global

On peut aussi définir un ordonnancement global, pour le corps de la fonction. Parmis les difficultés:

- Comment définir une instance d'exécution d'une instruction? (boucles!)
- Comment calculer les dépendences entre les instances d'exécution?
- Comment produire le code étant donné l'ordonnancement?

En réalité, il est plus simple d'ordonnancer directement le code source, en amont du compilateur.

Le modèle polyédrique fourni un cadre formel pour construire de tels ordonnancements sur des programmes réguliers (boucles `for`, tests et indices de tableaux prédictibles statiquement). cf. M2...