Optimisation II [1]
23 novembre 2011

1. Phases finales : allocation de registres et linéarisation
   - Graphe d'interférence et coloriage
   - Phase 4: Production du code LTL
   - Phase 5: Linéarisation du code
   - Phase 6: Génération de code assembleur

2. Autres optimisations
   - Blocs de base
   - Simplification des expressions
   - Optimisation de boucle

1. Ce cours s'inspire des chapitres 10 et 11 du cours de compilation de J.-C. Filliâtre (ENS)

Sommaire

Définition et usages

- Pour calculer les variables vivantes en un point de programme il faut préciser les variables utilisées et définies par chaque instruction.
- Le calcul est simple dans beaucoup de cas:

instruction	défini	utilisé
addi r1 r2 r3	r1	r2 r3
sle r1 r2 l		r1 r2
lw r1 n(r2)	r1	r2
sw r1 n(r2)		r1 r2
move r1 (r2)	r1	r2

- L'appel de fonctions est le cas délicat.

Définition et usages lors de l'appel de fonctions

- Les usages et définitions réalisées lors d'un appel de fonction dépendent du corps de cette fonction et comment il est compilé.
  Il est difficile de faire une analyse précise.
  On se contente d'approcher le résultat.
- Cas des appels à `syscall` pour **print** et **malloc**:
  Les registres $a0 et $v0 sont lus et le registre $v0 est redéfini.
- Cas d'un appel **call** f:
  - Les registres utilisés par les paramètres sont lus
  - Les registres caller-saved peuvent être librement utilisés par l'appelé - on les considère donc comme définis lors de l'appel.
- Cas de l'instruction **return**:
  - Instruction qui sort de la fonction
  - Pas de registres définis par cette instruction
  - A la sortie, les registres callee-saved en particulier $ra et $v0 peuvent être utilisés.

Graphe d'interférence

- Les nœuds du graphes sont les registres (physiques ou pseudo)
- Les arêtes correspondent à des interférences (impossible d'être dans le même registre physique ou à des préférences (être dans le même registre évite une instruction **move**)
- Si une instruction i définit le registre r et que les variables vivantes ($\texttt{out}(i)$ sont $r_1, \ldots, r_n$ on ajoute des arêtes d'interférence $r_i \leftrightarrow r_j$.
- Si l'instruction est un **move** r r_i alors on introduit une arête de préférence au lieu d'une arête d'interférence (sauf si une interférence existe déjà)

Allocation des registres

On ignore dans un premier temps les arêtes de préférence.

- Respecter les interférences est essentiel pour la correction du programme.
- Mettre des valeurs en registre accélère sensiblement les accès.
- Les préférences évitent des instructions **move** moins coûteuses.

Problème à traiter

- Déterminer une répartition des variables sur k-registres revient à trouver un k-coloriage du graphe.
- Problème NP-complet en général, on utilise une approximation.

Algorithme

- Si le graphe est vide, alors le coloriage est évident.
- Si le graphe comporte un nœud x de degré strictement inférieur à k,
  - on construit un nouveau graphe G' en retirant ce nœud et les arêtes correspondantes,
  - on colorie recursivement G'
  - on peut donner une couleur à x différente de celle de ses voisins
- Si le graphe ne comporte que des nœuds de degré supérieur à k alors on choisit un nœud x que l'on retire et on colorie le graphe résultant. On regarde le nombre de couleurs utilisées par les voisins de x
  - S'il en reste une de libre alors il est possible de colorier x et on l'ajoute au graphe.
  - Sinon, le coloriage a échoué

Gestion des échecs

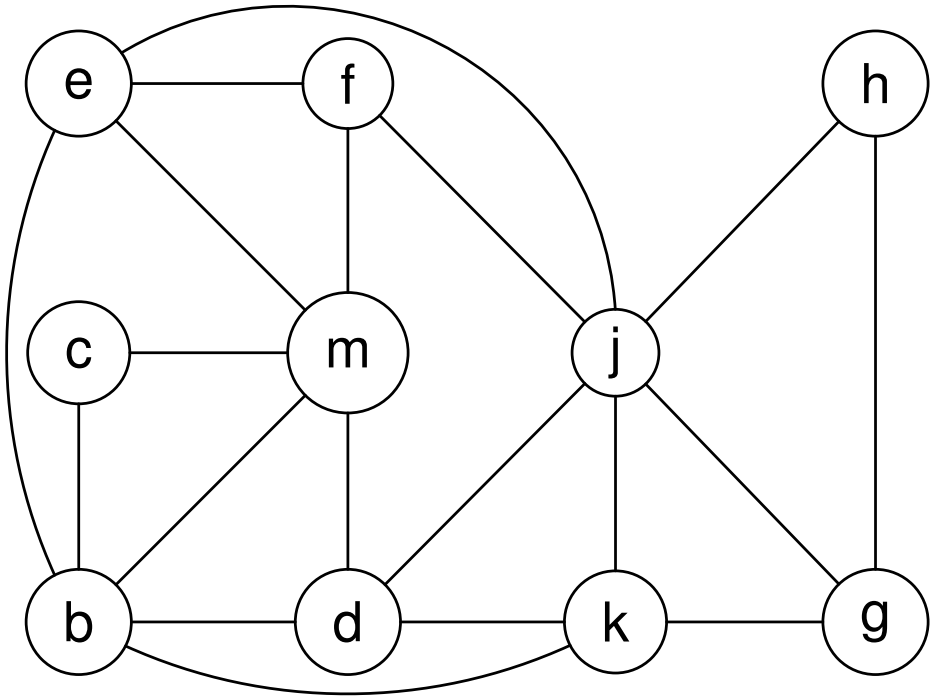
- Si le coloriage a échoué pour x, on décide de stocker x dans la mémoire.
- On choisit un emplacement mémoire m_x pour stocker x et on introduit pour chaque utilisation de x dans le programme une nouvelle variable x_i.
  - Si la valeur de x est utilisée dans une expression, on commencera par mettre dans x_i la valeur stockée à l'adresse m_x de la mémoire.
  - Si x est mise à jour alors on remplace x par x_i dans la mise à jour, et on fait suivre cette instruction d'une mise à jour de m_x dans la mémoire par la valeur x_i.
- La durée de vie des variables x_i est courte, elles n'interfèrent pas avec les autres variables.
- Il faut alors modifier le graphe d'interférence et recommencer le coloriage.

Exemple (d'après Appel)

Affectations simples et accès à la mémoire.
Les variables j, k doivent être définies avant le programme.
Les variables d, k, j sont utilisées après cette portion de programmes.

instruction		var. vivantes
lw	g 12(j)	j,k
addi	h k -1	j,g,k
mul	f g h	j,g,h
lw	e 8(j)	j,f
lw	m 16(j)	e,j,f
lw	b f	e,m,f
addi	c e 8	e,m,b
move	d c	c,m,b
addi	k m 4	d,m,b
move	j b	d,k,b
		d,k,j

Graphe d'interférence

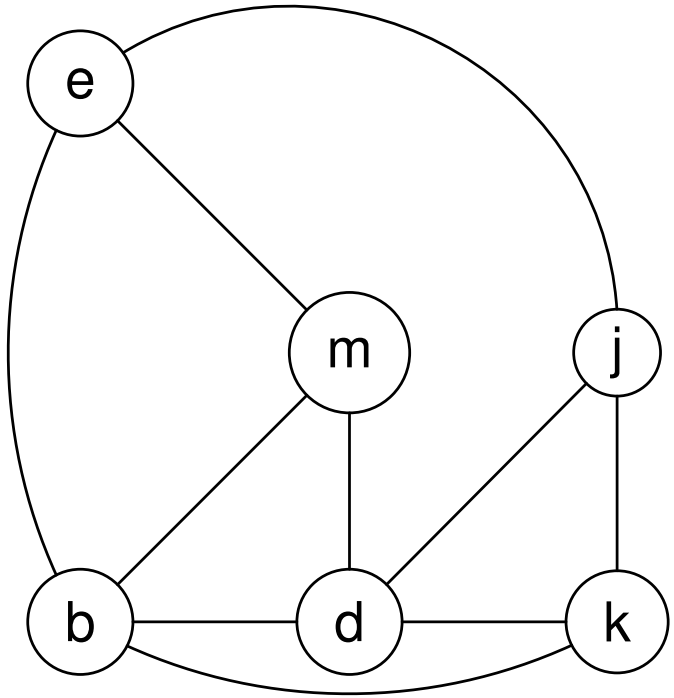


Degrés des noeuds:

b:	5	c:	2	d:	4	e:	4	f:	3
g:	3	h:	2	j:	6	k:	4	m:	5

Coloriage avec 4 registres

On enlève des nœuds de degré inférieur à 3: *g*, *h*, *c*, *f*.

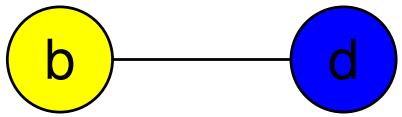


Degrés des noeuds:

b:	4	d:	4	e:	3
j:	3	k:	3	m:	3

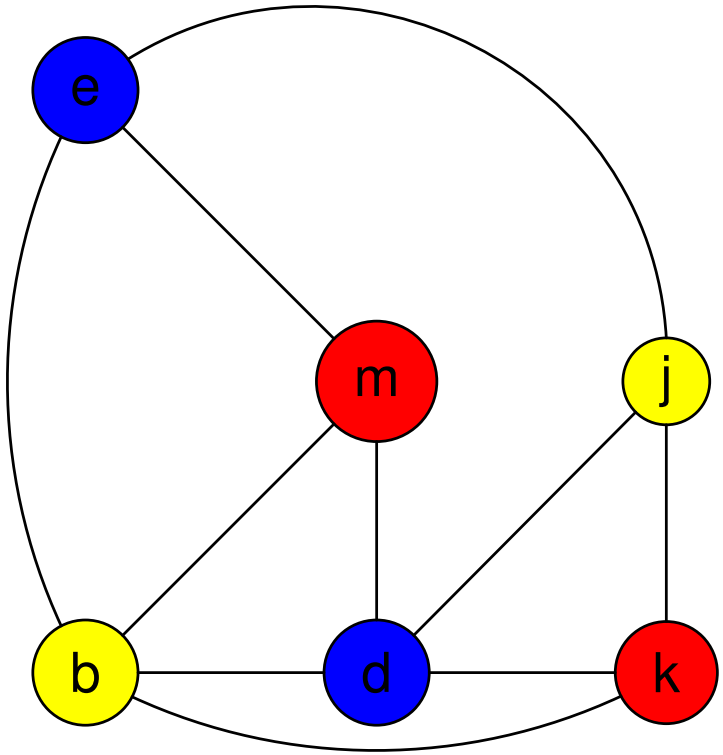
Coloriage - suite

On enlève des nœuds de degré inférieur à 3: j, k, e, m.
On peut colorier les nœuds restants.



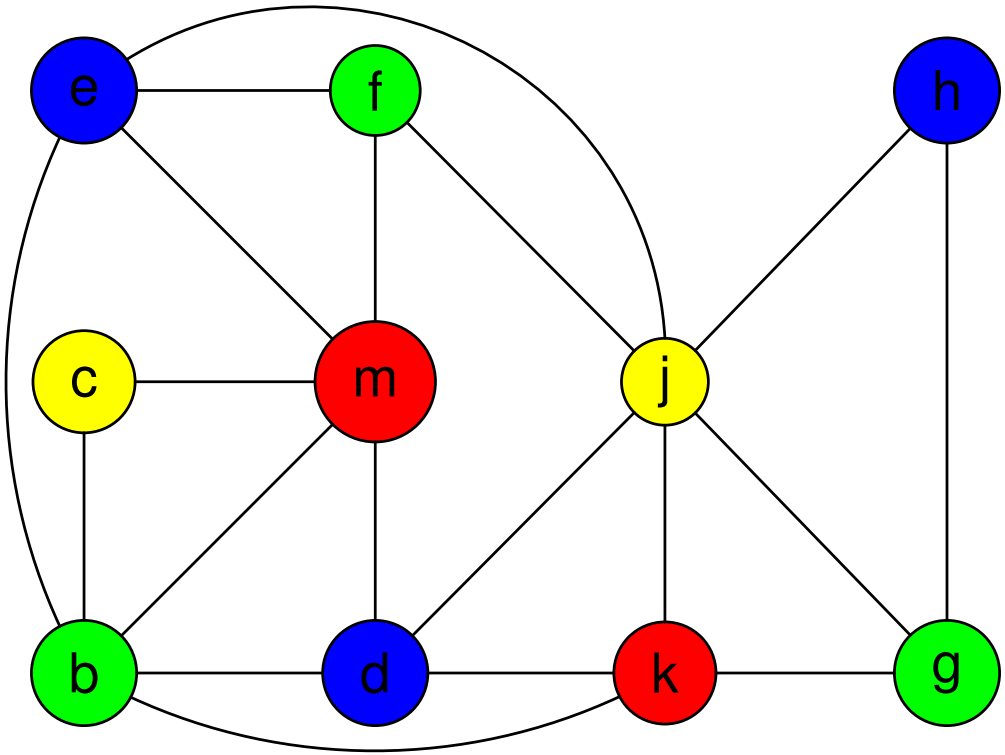
Coloriage - suite

On réintroduit les noeuds: *j*, *k*, *e*, *m*.



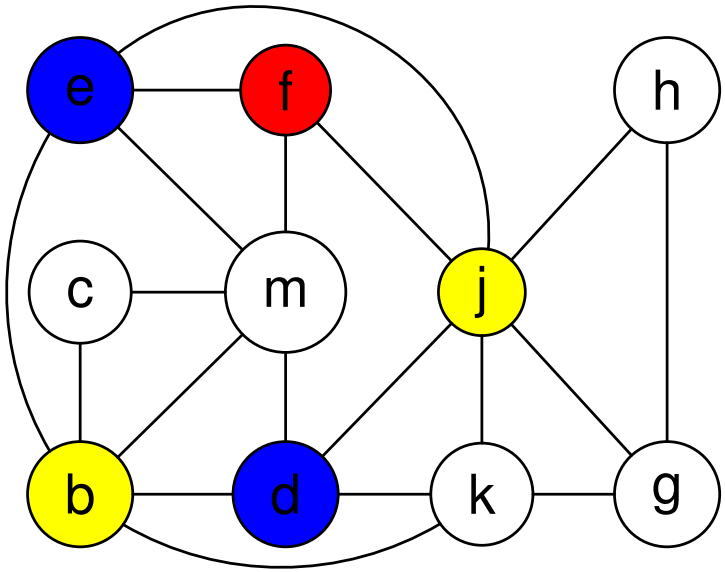
Coloriage - suite

On réintroduit les noeuds: *g*, *h*, *c*, *f*.



Coloriage avec 3 registres

- On peut refaire l'analyse en examinant les nœuds dans l'ordre suivant. $b, d, j, e, f, \mathbf{m}, \mathbf{k}, g, c, h$
- Les nœuds indiqués en gras sont ceux dont le degré est supérieur à 3 et donc qui peuvent poser un problème.
- Une tentative de coloriage b jaune, d bleu, j jaune, e bleu, f rouge ne permet pas de colorier m.



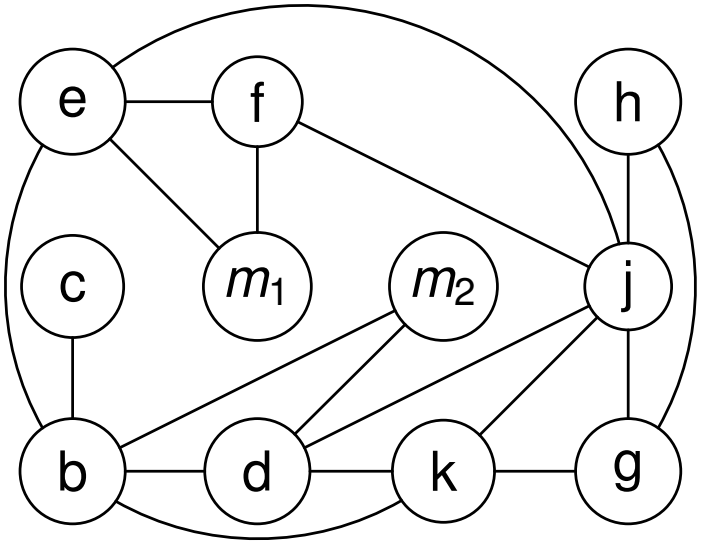
Transformation

- On stocke m en mémoire à l'adresse `M`.
- Chaque accès à m se fait par l'intermédiaire d'une nouvelle variable m_i

lw	g 12(j)	j,k
addi	h k -1	j,g,k
mul	f g h	j,g,h
lw	e 8(j)	j,f
lw	m1 16(j)	e,j,f
sw	m1 M	e,m1,f
lw	b f	e,f
addi	c e 8	e,b
move	d c	c,b
lw	m2 M	b,d
addi	k m2 4	d,m2,b
move	j b	d,k,b
		d,k,j

Coloriage avec 3 registres

- Le nœud m de degré 5 s'est transformé en deux nœuds m_1 et m_2 de degré 2.

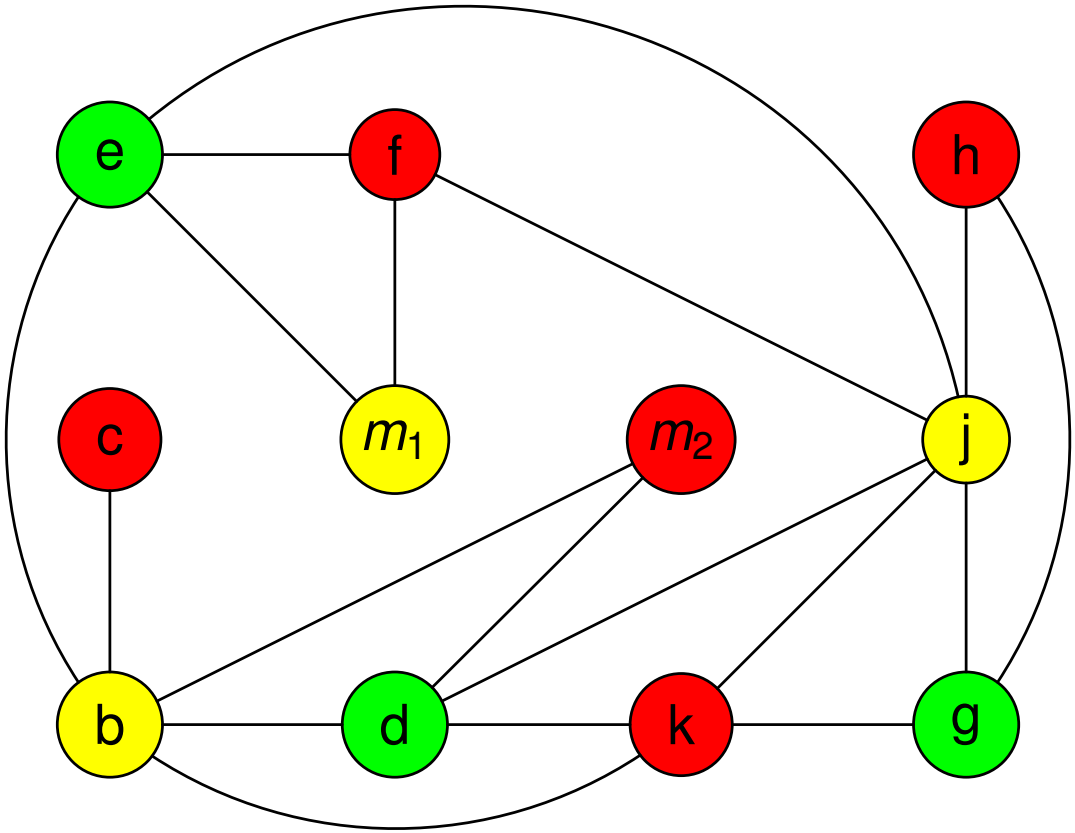


- On peut réexaminer les nœuds du nouveau graphe dans l'ordre suivant :

$$j, g, k, d, b, e, f, m_1, m_2, c, h$$

- Il n'y a plus de nœud à problème.

Résultat



Coloriage et allocation de registres

Cas général

- La couleur d'un nœud correspond à un registre physique utilisable pour stocker la valeur.
- Le graphe d'interférence comporte quelques registres physiques: $\$a_i$ et $\$v_0$. Ils sont supposés être déjà coloriés de la couleur correspondante.
- Le graphe est représenté comme une map qui associe à chaque registre:
  - L'ensembles des registres en préférence
  - L'ensemble des registres en interférence
  - L'ensemble (mutable) des registres/couleurs possibles
- Les couleurs sont initialisées par les registres physiques possibles qui ne sont pas en interférence.

Heuristique simple de coloriage

Recherche d'un nœud à colorier

1. un nœud qui a exactement une couleur qui est une préférence
2. un nœud qui a une seule couleur possible
3. un nœud qui a une préférence dont la couleur a déjà été choisie
4. un nœud qui a encore une couleur possible
5. sinon on choisit un nœud qui sera stocké sur la pile.

Coloriage

Tant qu'il reste des nœuds à colorier:

- choisir si possible un nœud et une couleur
- retirer la couleur choisie des couleurs possibles des nœuds qui interfèrent
- retirer le nœud et continuer
- s'il n'y a plus de coloriage possible, on décide de mettre les nœuds restant dans la pile.

Registres sauvegardés dans la pile

- Les accès à des pseudo-registres sauvegardés dans la pile demandent au moins 1 regitre pour les transferts (2 dans le cas d'opération binaire).
- On peut réintroduire des pseudo-registres pour ces accès, il faut alors recommencer le coloriage (utile sur des architectures à peu de registres).
- On peut aussi réserver deux registres a priori pour cet usage.
- On peut utiliser le graphe d'interférence pour économiser l'espace dans la pile pour stocker les registres: les couleurs seront alors les positions dans la pile, en nombre non borné que l'on cherchera à minimiser.

Sommaire

Code LTL

- Le coloriage nous fournit une adresse physique pour chaque pseudo registre, ou bien un décalage dans la pile:

  ```
  type color = Spilled of int | Reg of Register.t
  ```

  Deux regitres physiques `tmp1` et `tmp2` permettent l'accès dans la pile.
- On connait pour chaque graphe le nombre de registres à stocker sur la pile, on peut en déduire la taille du tableau d'activation.
- Les instructions d'accès aux paramètres deviennent des accès dans la pile par rapport au sommet de pile $\$sp$
- Les instructions d'allocation/suppression du tableau d'activation disparaissent.

Accès aux pseudo-registres

Fonctionnelles pour accéder aux pseudo registres:
Si le pseudo-registre *r* est stocké dans la pile:

- écrire dans *r*: écrire dans *tmp1* puis mettre à jour la pile.
- lire dans *r*: mettre la valeur de la pile dans *tmp1* puis utiliser *tmp1*.

```
(* c coloriage, registre machine pour la variable r *)
let lookup c r = if is_hw r then Reg r else M.find r c

let write c r l = match lookup c r with
  | Reg hr -> hr, l
  | Spilled n -> tmp1, add_instr (Eset_stack (tmp1, n, l))

let read1 c r f = match lookup c r with
  | Reg hr -> f hr
  | Spilled n -> Eget_stack (tmp1, n, add_instr (f tmp))
```

Production de code LTL

```
let ltl_instr c frame_size = function
...
 |ErtlEGaccess (r,x,l) ->
      let hr, l = write c r l in
      EGaccess (hr, x, l)
 |ErtlEGassign (r,x,l) ->
      read1 c r (fun hw -> EGassign (hw, x, l))
 |ErtlEmove(r1,r2,l) ->
   begin match lookup c r1, lookup c r2 with
    | w1, w2 when w1 = w2 -> Egoto l
    | Reg hr1,Reg hr2       -> Emove(hr1,hr2,l)
    | Reg hr1,Spilled of2 -> Eget_stack(hr1,of2,l)
    | Spilled of1,Reg hr2 -> Eset_stack(hr2,of1,l)
    | Spilled of1,Spilled of2 ->
      Eget_stack(tmp1,of2,add_instr(Eset_stack(tmp1,of1,l))
   end
...
```

Production de code LTL

```
let ltl_instr c frame_size = function
...
  | Ertl.Ecall (x,n,l) ->  Ecall (x,l)
  | Ertl.Ealloc_frame l
  | Ertl.Edel_frame l when frame_size = 0 -> Egoto l
  | Ertl.Ealloc_frame l ->
      Eunop(Register.sp,Addi (- frame_size),Register.sp, l)
  | Ertl.Edel_frame l ->
      Eunop(Register.sp,Addi frame_size, Register.sp,l)
  | Ertl.Eget_param (r,n,l) ->
      let hwr, l = write c r l in
      Eget_stack(hwr,frame_size+n,l)
  | Ertl.Eset_param (r, n, l) ->
      read1 c r (fun hwr -> Eset_stack (hwr,n,l))
...
```

Sommaire

LIN: code linéarisé

- Réécrire le code de manière linéaire avec des instructions qui se suivent en minimisant les instructions de saut.
- Parcours du graphe en suivant les instructions dans l'ordre d'exécution et en mémorisant dans une table `visited` les étiquettes des nœuds visités.
- Dans un premier temps, chaque instruction peut être la cible d'un branchement et garde donc son étiquette.
- On mémorise dans une autre table `label` les étiquettes utiles dans le programme assembleur (cibles d'instructions de saut)
- Implantation sous la forme d'une fonction `lin` *g l*: *g* est le graphe du code, *l* est une étiquette où se rendre et à partir de laquelle on veut linéariser le code.

Algorithme de linéarisation

- Si l est déjà visitée alors on produit l'instruction **b** l; l'étiquette l est utile.
- sinon on mémorise la visite de l, soit i l'instruction associée à l dans g
- Si i est une instruction simple qui mène à l'étiquette l'
  - on produit l'instruction associée à i
  - on continue de linéariser à partir de l'
- Si i est une instruction de branchement qui mène à $l1$ et $l2$
  - Si $l1$ et $l2$ non visités:
    - on produit l'instruction de branchement correspondant à i pour aller à $l1$;
    - $l1$ est une étiquette marquée utile;
    - on linéarise le code à partir de $l2$.
    - on linéarise le code à partir de $l1$.
  - Si $l1$ déjà visitée:
    - on produit l'instruction de branchement correspondant à i pour aller à $l1$;
    - $l1$ est une étiquette marquée utile;
    - on linéarise le code à partir de $l2$.
  - Si $l2$ déjà visitée mais pas $l1$: on inverse le test dans l'instruction i et on se ramène au cas précédent.
- Si i est une instruction **goto** l', on introduit l'étiquette l et on linéarise à partir de l'.

Sommaire

Production du code assembleur

- Peut se faire en même temps que la linéarisation ou après.
- Les variables globales, les constantes correspondant à des chaînes de caractères doivent être collectées et déclarées dans la zone de données.
- Traductions des instructions de LTL vers l'assembleur MIPS

LTL	MIPS
`Ereturn`	**jr** $*ra*
`Ecall` *f*	**jal** *f*
`EGaccess` $*r* *x*	**lw** $*r* *x*
`EGassign` $*r* *x*	**sw** $*r* *x*
`Eget_stack` $*r* *n*	**lw** $*r* *n*($*sp*)
`Eset_stack` $*r* *n*	**sw** $*r* *n*($*sp*)

Conclusion sur les optimisations

- Une bonne utilisation des registres est une étape cruciale de l'optimisation des programmes.
- Passage par une forme intermédiaire qui suppose une infinité de pseudo-registres et créé de nombreuses copies inutiles.
- Stratégie de rangement des pseudo-registres dans la mémoire physique.
- Algorithmique des graphes.

Sommaire

Blocs de base

Pour effectuer des optimisations, il est utile d'organiser le graphe de flots de contrôle en blocs de base.

- Le graphe de flots de contrôle a comme nœuds les instructions et une arête si une instruction peut en suivre une autre.
- Un bloc est une suite d'instructions consécutives maximale telle que
  - le flôt de contrôle ne peut rentrer que par la première instruction (et pas au milieu du bloc).
  - le contrôle ne peut quitter le bloc par un branchement qu'éventuellement à la dernière instruction.
- Les blocs forment les nœuds du graphe de flots de contrôle par blocs.

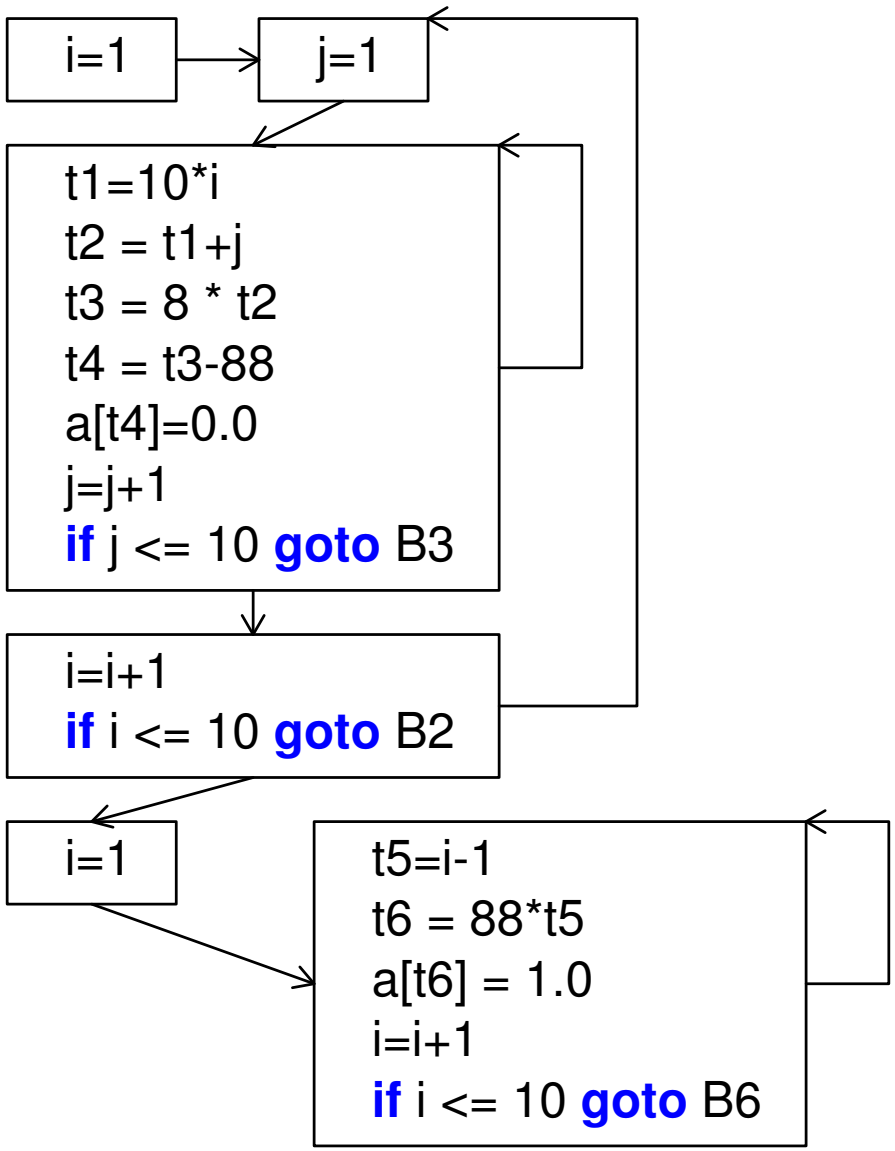
Algorithme pour trouver les blocs

- On détermine les instructions de tête des blocs (initialement le point d'entrée dans le programme).
- Toute instruction qui est la cible d'un branchement conditionnel ou inconditionnel est une tête de bloc
- Toute instruction qui suit un branchement conditionnel ou inconditionnel est une tête de bloc.
  Dans le cas d'un branchement inconditionnel, si l'instruction suivante n'est pas la cible d'un branchement alors c'est du code mort.
- Les blocs sont formés de la suite d'instructions entre deux têtes de bloc.

Exemple

```
    i=1
L2:j=1
L3:t1=10*i
    t2 = t1+j
    t3 = 8 * t2
    t4 = t3-88
    a[t4]=0.0
    j=j+1
    if j <= 10 goto L3
    i=i+1
    if i <= 10 goto L2
    i=1
L13:t5=i-1
    t6 = 88*t5
    a[t6] = 1.0
    i=i+1
    if i <= 10 goto L13
```

Graphe par blocs



Optimisations par bloc

- A l'intérieur d'un bloc il est possible de réordonner les opérations afin de procéder à des optimisations.
- Reconnaissance de sous-expressions locales
- Utilisation d'identité algébriques
- Identification de code mort
- Réorganisation des instructions

Sommaire

Reconnaissance de sous-expressions communes

- On peut reconnaître qu'une même expression a déjà été calculée et réutiliser la valeur.
- En présence d'alias (expressions différentes pouvant représenter la même place mémoire) comme les tableaux ou les pointeurs, il faut veiller que certaines affectations peuvent invalider des calculs faits précédemment.
- Les égalités algébriques doivent être utilisées avec précaution car certaines ne sont pas vérifiées par l'arithmétique des ordinateurs ou ne préservent pas l'ordre des calculs:

$$x < y \Leftrightarrow 0 < y - x$$

Représentation des expressions par des DAG

DAG : directed acyclic graph (comme un arbre mais avec partage de sous arbres)

- Une technique classique dite de “hash-consing” pour une représentation efficace (en mémoire) de grosses expressions.
- Entrée : un “terme” (par exemple une expression arithmétique) représenté sous forme arborescente.
- Sortie : Le même terme représenté sous forme de DAG (deux sous expressions égales auront la même représentation physique)
- Principe : une table garde une trace de tous les termes rencontrés.

Exemple

Type logique

```
type term =
    Var of ident | Cte of int | Op of op * term * term
```

Représentation concrète

```
type term_node =
    Var of ident | Cte of int | Op of op * term * term
and term = {node : term_node; tag : int}
```

- Le tag représente de manière unique le terme.
- La fonction de hash-consing prend en argument un objet de type `term_node` et renvoie un terme de type `term`
- Elle utilise une table de hachage qui associe à chaque `term_node` déjà rencontré, le terme “hash-consé” (de type `term` associé.

Table de hachage

Module `Hashtbl` avec fonction d'égalité spécifique.

```
module type HashedType = sig
    type t
    val equal : t -> t -> bool
    val hash : t -> int
end

module Make : functor (H : HashedType) ->  sig
    type key = H.t
    type 'a t = 'a Hashtbl.Make(H).t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val add : 'a t -> key -> 'a -> unit
    val find : 'a t -> key -> 'a
    val mem : 'a t -> key -> bool
    val fold : (key->'a->'b->'b) -> 'a t -> 'b -> 'b
 .... end
 val hash : 'a -> int
```

Mise en œuvre sur les termes

- Les clés sont les objets de type `term_node`
- Les objets stockés dans la table sont de type `term`
- La fonction d'égalité sur les clés ne doit pas être trop coûteuse
- Elle utilise l'égalité des tags pour les sous-termes.

```
let eq_term t1 t2 = match (t1,t2) with
  Var x1, Var x2 -> x1=x2
| Cte c1, Cte c2 -> c1=c2
| Op (op1,e1,f1), Op(op2,e2,f2)
            -> op1=op2 & e1.tag=e2.tag & f1.tag=f2.tag
| _ -> false
module H : HashedType = struct
    type t = term_node
    let equal = eq_term let hash = Hashtbl.hash
                                end
module HashTable = Hashtbl.Make(H)
```

Fonction de hash-consing

```
let (hashcons : term_node -> term) =
 let table = HashTable.create 251
 and tagref = ref 0
 in fun x -> try Hashtbl.find table x
             with Not_found ->
             let hc = { node = x; tag = !tagref} in
             incr tagref; HashTable.add table x hx; hx
```

On peut ensuite définir des fonctions de construction :

```
let (var:ident->term) x = hashcons (Var x)
let (cte:int->term) x = hashcons (Cte x)
let (op:op->term->term->term) o e1 e2
                        = hashcons (Op(o,e1,e2))
```

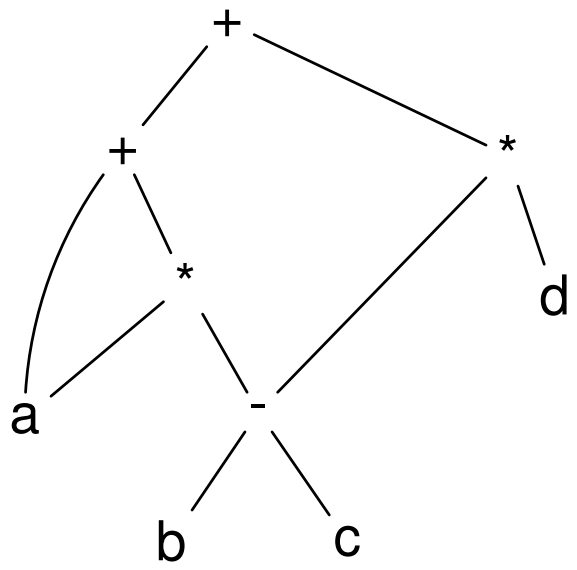
Invariant: $x = y \Leftrightarrow x == y \Leftrightarrow x.tag = y.tag$

Génération de code

- Pour engendrer le code intermédiaire pour une expression représentée sous forme de DAG, on peut associer une variable à chaque tag utilisé dans l'expression (ou la liste d'expressions) et engendrer le code à trois valeurs correspondant.
- On peut aussi exploiter des propriétés arithmétiques: lorsqu'on traite le terme $a + b$ ou $a \times b$ on peut chercher si $b + a$ ou $b \times a$ ont déjà été entrés et réutiliser les valeurs.

Exemple

- Expression $a + a * (b - c) + (b - c) * d$
- DAG



Génération de code

- Terme

```
{node=Var"a";tag=0}
{node=Var"b";tag=1}
{node=Var"c";tag=2}
{node=Op("-",{tag=1;..},{tag=2;..});tag=3}  (*b-c*)
{node=Op("*",{tag=0;..},{tag=3;..});tag=4}  (*a*(b-c)*)
{node=Op("+",{tag=0;..},{tag=4;..});tag=5}  (*a+a*(b-c)*)
{node=Var"d";tag=6}
{node=Op("*",{tag=3;..},{tag=6;..});tag=7}  (*(b-c)*d*)
{node=Op("+",{tag=5;..},{tag=7;..});tag=8}
```

- Code

```
r0 = a          r5 = r0+r4
r1 = b          r6 = d
r2 = c          r7 = r3*r6
r3 = r1-r2      r8 = r5+r7
r4 = r0*r3
```

Cas des instructions

- On part d'une suite d'instructions à 3 registres
- On cherche à partager des sous-expressions
- Exemple
  a = b+c
  b = a-d
  c = b+c
  d = a-d
- Les variables sont modifiables : les deux sous-expressions $b + c$ n'ont pas la même valeur.
- On construit le DAG correspondant aux expressions calculées et un environnement qui dit pour chaque variable quel est le terme correspondant.
- On suppose données les valeurs initiales des variables (contexte).
- On calcule librement les expressions puis on met à jour l'environnement.

Algorithme

- Pour chaque variable a utilisée, hash-conser le terme $\texttt{Var}\ "a"$ et associer dans l'environnement le terme obtenu à a.
- Pour une instruction $a = b\ \texttt{op}\ c$:
  - chercher les termes e_b et e_c associés à b et c;
  - hash-conser le terme $\texttt{Op}(op, e_b, e_c)$ et associer dans l'environnement ce terme à la variable a.
- On calcule librement les expressions dans le dag à l'aide de registres.
- On met à jour à la fin les variables réutilisées dans la suite.

Traitement de tableaux

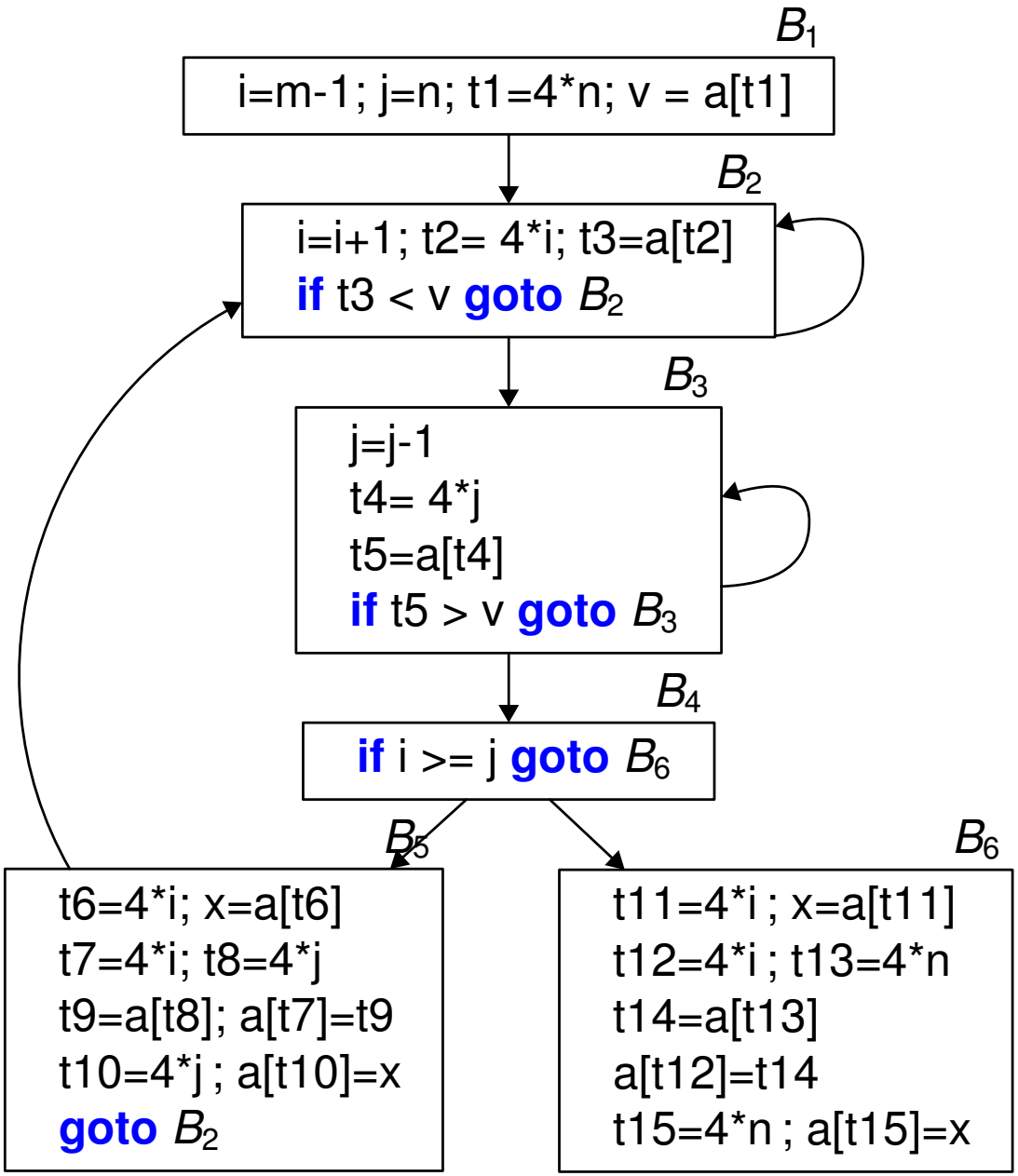
- Deux instructions $x = a[i]$ et $a[i] = y$.
- a est une variable représentant une adresse mémoire.
- Problème: x=a[i]
  a[j]=y
  z=a[i]
- Plusieurs expressions $a[i]$ et $a[j]$ peuvent représenter le même emplacement mémoire (alias).
- On associe à la variable a un ensemble de termes calculés valides.
- Lorsqu'une affectation $a[i] = y$ est traitée, l'ensemble est vidé.
- Lorsqu'une affectation $x = a[i]$ est traitée, on hash-conse le terme $a[i]$.
  - si ce terme existe, on vérifie qu'il est valide
  - si le terme n'existe pas ou est invalide, on le crée et on ajoute le terme crée à la liste des valides de a, on associe également le terme obtenu à la variable x dans l'environnement.

Sommaire

Optimisation de boucles

- Beaucoup de programmes passent une grande partie de leur temps dans des boucles internes.
- Intérêt d'optimiser ces boucles.
  - Faire sortir des instructions constantes
  - Eviter des calculs redondants

Exemple


B_1
i=m-1; j=n; t1=4*n; v = a[t1]
B_2
i=i+1; t2= 4*i; t3=a[t2]
if t3 < v goto B_2
B_3
j=j-1
t4= 4*j
t5=a[t4]
if t5 > v goto B_3
B_4
if i >= j goto B_6
B_5
t6=4*i; x=a[t6]
t7=4*i; t8=4*j
t9=a[t8]; a[t7]=t9
t10=4*j ; a[t10]=x
goto B_2
B_6
t11=4*i ; x=a[t11]
t12=4*i ; t13=4*n
t14=a[t13]
a[t12]=t14
t15=4*n ; a[t15]=x

Elimination de sous-expressions communes

On peut simplifier les blocs B_5 et B_6
($t6 = t7, t11 = t12, t11 = t12, t13 = t15$)

B_5 devient:

```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
```

B_6 devient:

```
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=t14
a[t13]=x
```

On peut aussi faire des optimisations globales pour reprendre les calculs de $t1 = 4 * n$, $t2 = 4 * i$ et $t4 = 4 * j$ effectués dans B_1, B_2 et B_3:

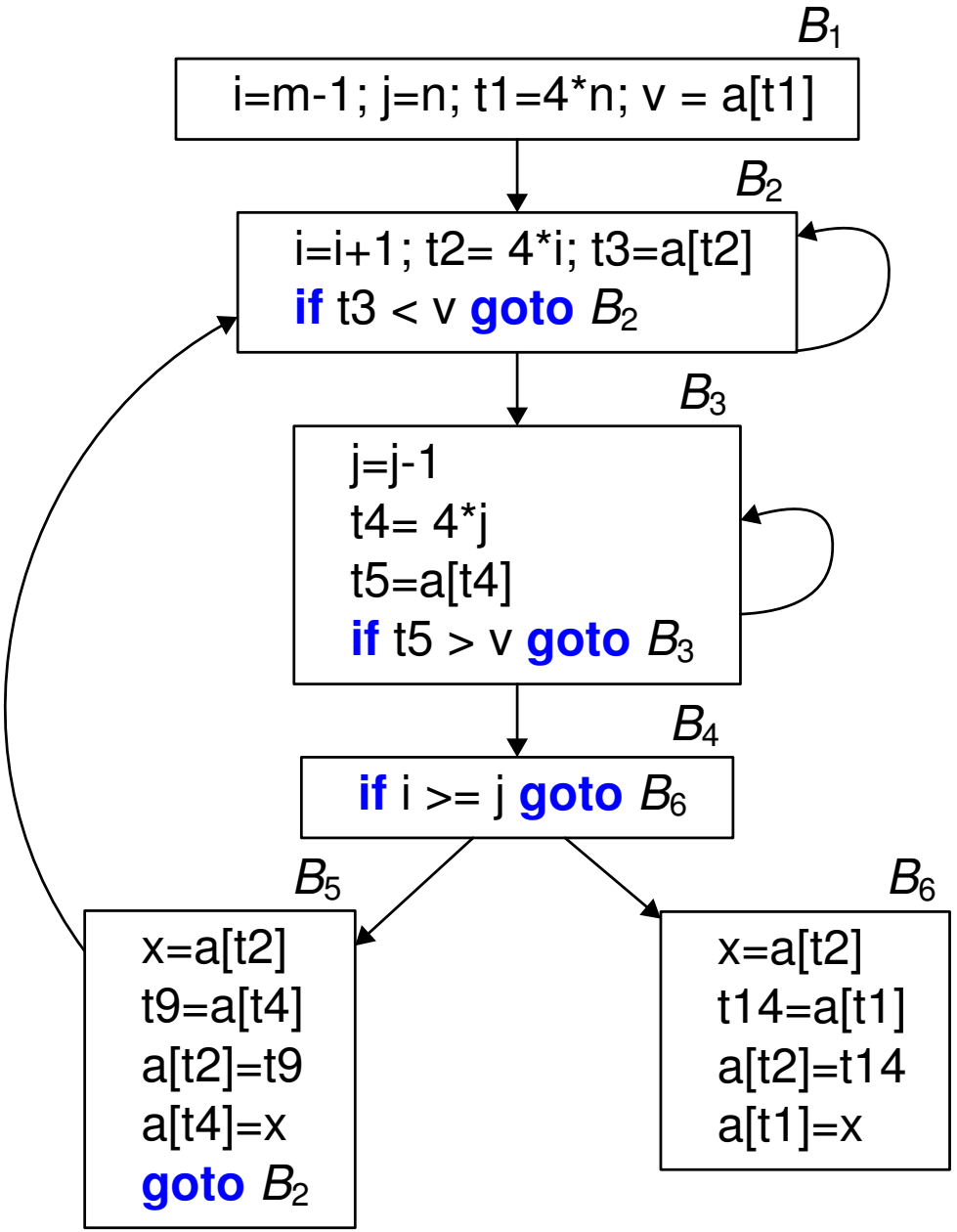
B_5 devient:

```
x=a[t2]
t9=a[t4]
a[t2]=t9
a[t4]=x
```

B_6 devient:

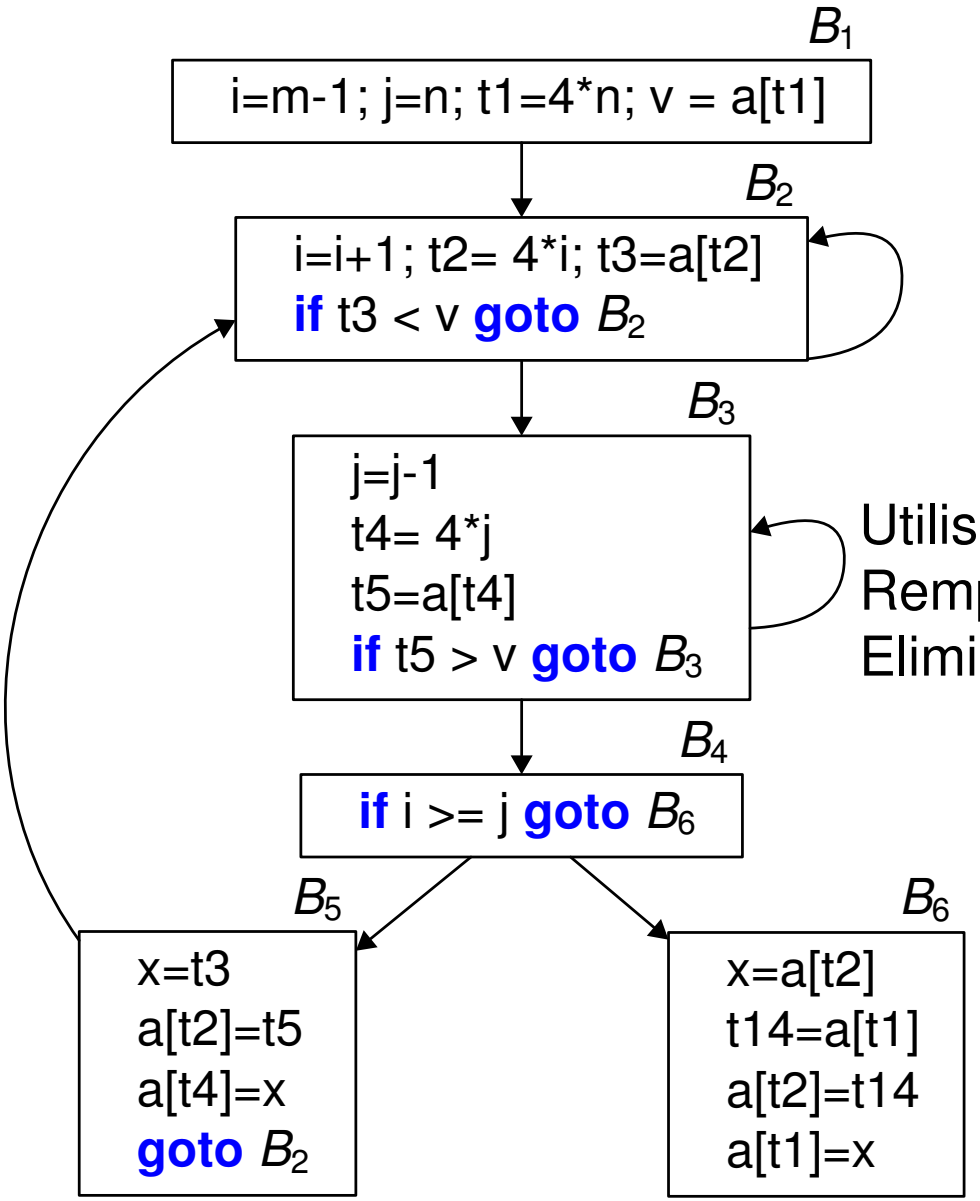
```
x=a[t2]
t14=a[t1]
a[t2]=t14
a[t1]=x
```

Résultat



a[*t2*] calculé dans *t3*
pas modifié ensuite
a[*t4*] calculé dans *t5*
pas modifié ensuite
a[*t1*] calculé dans *v*
mais peut être modifié

Résultat



Utilisation de la copie $x = t3$
Remplacer x par $t3$ si possible
Eliminer x si non active

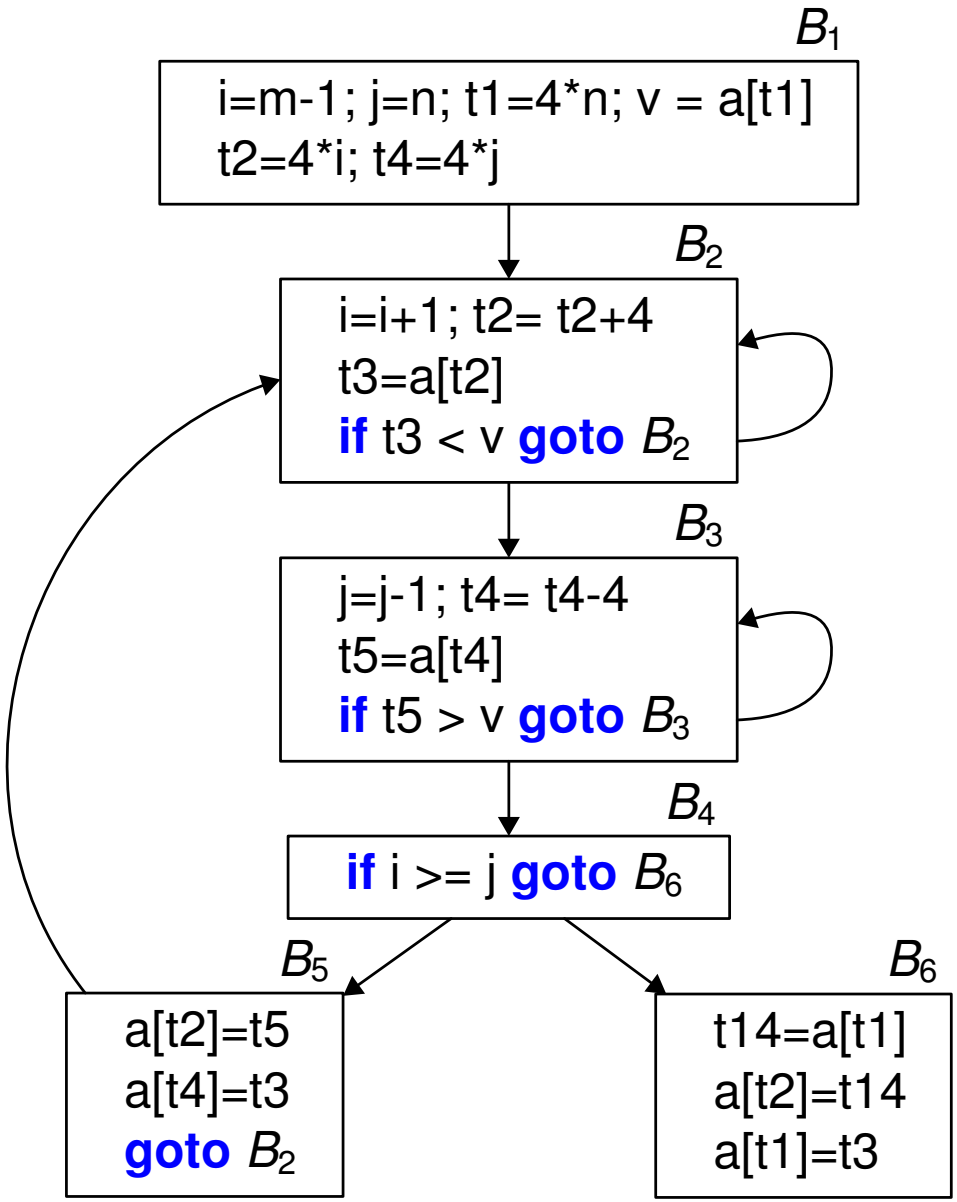
Notion de boucle

- Une boucle est un ensemble L de blocs dans le graphe de contrôle tel que
  - Il y a un seul bloc d'entrée B_0 dans L qui a un prédécesseur hors de L
  - Pour tout bloc B de L, il existe un chemin de B à B_0.
    (Il y a également un chemin de B_0 à B sinon B ne serait pas accessible.
- Exemple : $L = \{B_2\}$, $L = \{B_3\}$ sont des boucles ainsi que $L = \{B_2, B_3, B_4, B_5\}$.
- Données intéressantes sur les boucles:
  - invariant de boucle: expression qui prend la même valeur à toutes les itérations de boucle et qui pourra être calculée avant le bloc de base.
  - variable d'induction: variable qui à chaque passage dans la boucle augmente d'une valeur constante c.
    On utilise des opérations d'incrémentation moins coûteuses que les multiplications.
    Si plusieurs variables d'induction sont corrélées, on en garde une seule.

Exemple

- Dans la boucle B_2 on effectue les opérations $i = i + 1; t2 = 4 * i$.
- Les variables i et $t2$ ne sont pas modifiées par ailleurs.
- Il suffit d'initialiser $t2$ une fois, puis d'incrémenter $t2$ de 4 à chaque itération.
- On peut faire de même pour j et $t4$ dans le bloc B_3.

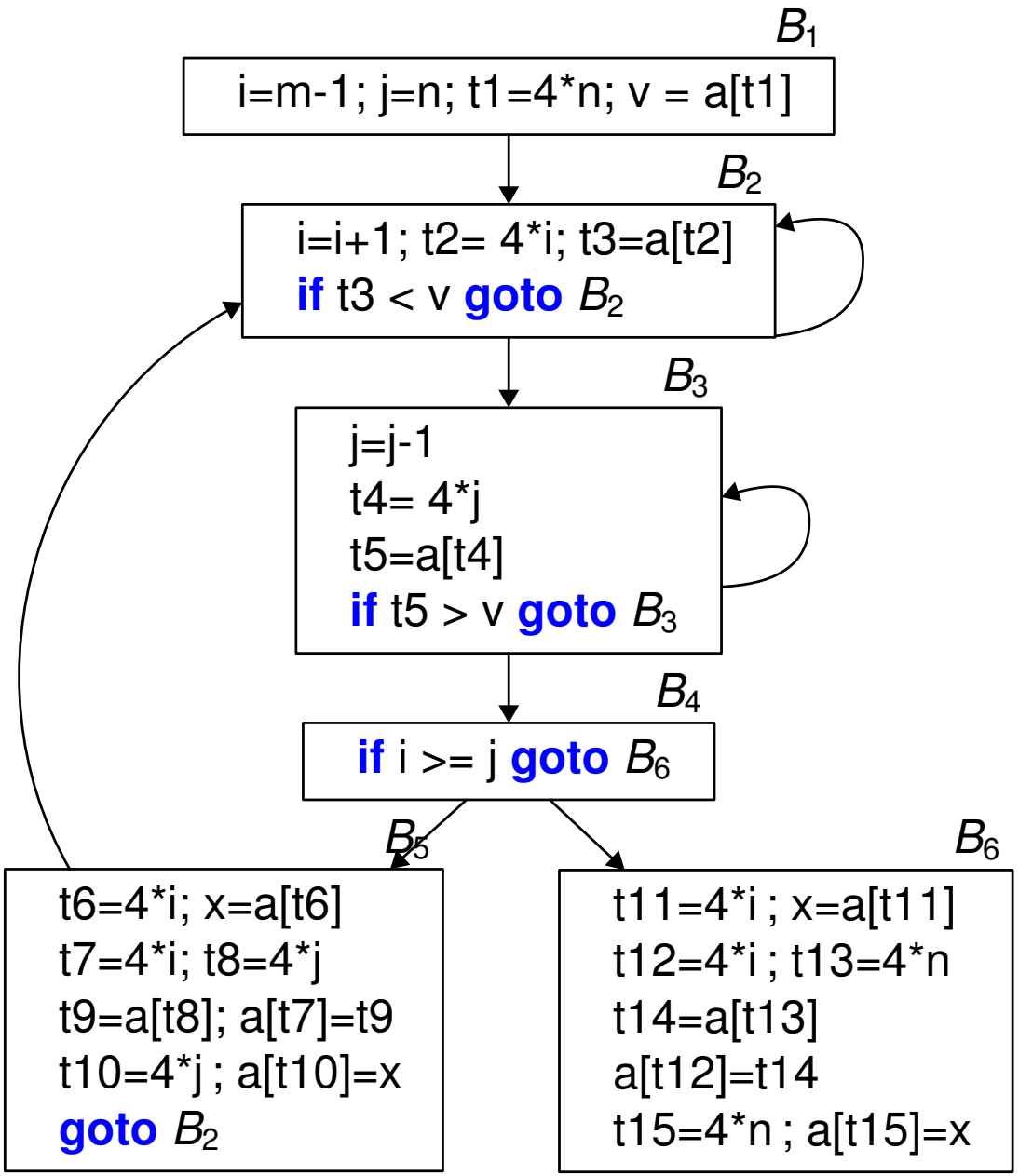
Résultat

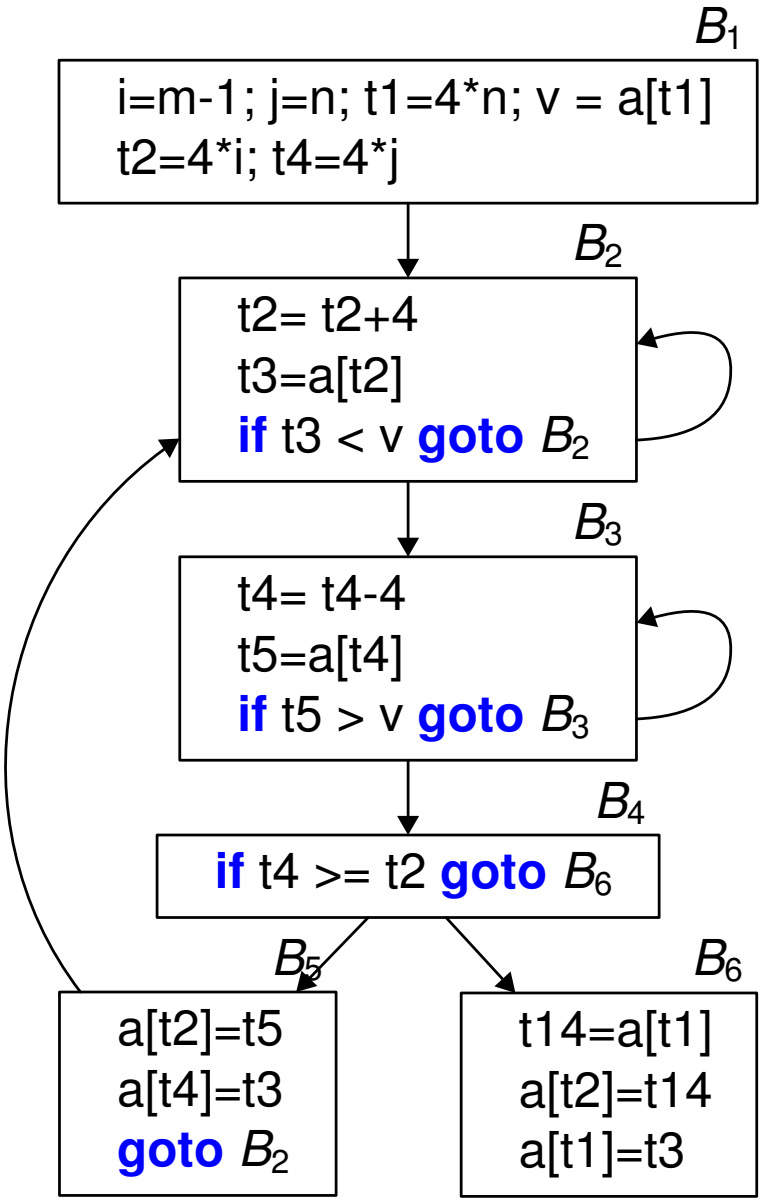

B_1
i=m-1; j=n; t1=4*n; v = a[t1]
t2=4*i; t4=4*j
B_2
i=i+1; t2= t2+4
t3=a[t2]
if t3 < v goto B_2
B_3
j=j-1; t4= t4-4
t5=a[t4]
if t5 > v goto B_3
B_4
if i >= j goto B_6
B_5
a[t2]=t5
a[t4]=t3
goto B_2
B_6
t14=a[t1]
a[t2]=t14
a[t1]=t3

Simplification supplémentaire

- On analyse ensuite la boucle B_2, B_3, B_4, B_5
- Les variables i et j ne sont utilisées que pour le test $i \geq j$
- La variable i est corrélée à $t2 = 4 * i$ et la variable j est corrélée à $t4 = 4 * i$.
- Le test $i \geq j$ est équivalent à $t2 \geq t4$.
- On peut alors éliminer les variables i et j dans les autres blocs que B_1.

Résultat transformation du code





Conclusion sur les optimisations

- Les optimisations sont des opérations complexes qui peuvent ralentir sensiblement la compilation.
- Les optimisations doivent préserver la sémantique des programmes.
- Les optimisations peuvent avoir des effets divers sur la taille du code et sa vitesse d'exécution.
- Les compilateurs proposent différentes options pour utiliser ou non certaines classes d'optimisations.
- Les optimisations en accroissant la distance entre le code source et le code exécuté peuvent rendre difficile voir impossible l'utilisation de debugueurs.