Allocation de registres
Coloriage de graphe

Luc Maranget

2004

Place dans la chaîne de compilation

$$\mathrm{SA}\ \overbrace{\xrightarrow{\text{Générat. Canon.}}}^{(1)}\ \mathrm{CI}\ \overbrace{\xrightarrow{\text{Sélection d'instruct.}}\ \mathrm{CA}\ \underbrace{\boxed{\xrightarrow{\text{Durée de vie}}}\ \mathrm{CA}\ \xrightarrow{\text{Coloriage}}}_{\text{Allocation de registres}}}^{(2)}\ \mathrm{CA}$$

Rappel le code assembleur est *presque* de l'assembleur : il contient des temporaires.

Le but de l'allocation de registres est de transformer ces temporaires en registres de la machine.

Allocation en pile ou *spill*

On ne peut pas toujours transformer les temporaires en registres :

- La *pression* est trop forte.
- Certains temporaires interfèrent avec tous les registres.

On doit alors *spiller* des temporaires.

```
add t1, t2, 2
mul t1, t1, t2
```

```
lw  $t8, 4($sp) # load t2
add $t8, $t8, 2
sw  $t8, 0($sp) # store t1
lw  $t8, 0($sp) # load t1
lw  $t9, 4($sp) # load t2
mul $t8, $t8, $t9
sw  $t8, 0($sp) # store t1
```

Spill avec éphémères

La fonction `Spill.spill_fun` réécrit le code et alloue des cases dans le frame (zone des locaux).

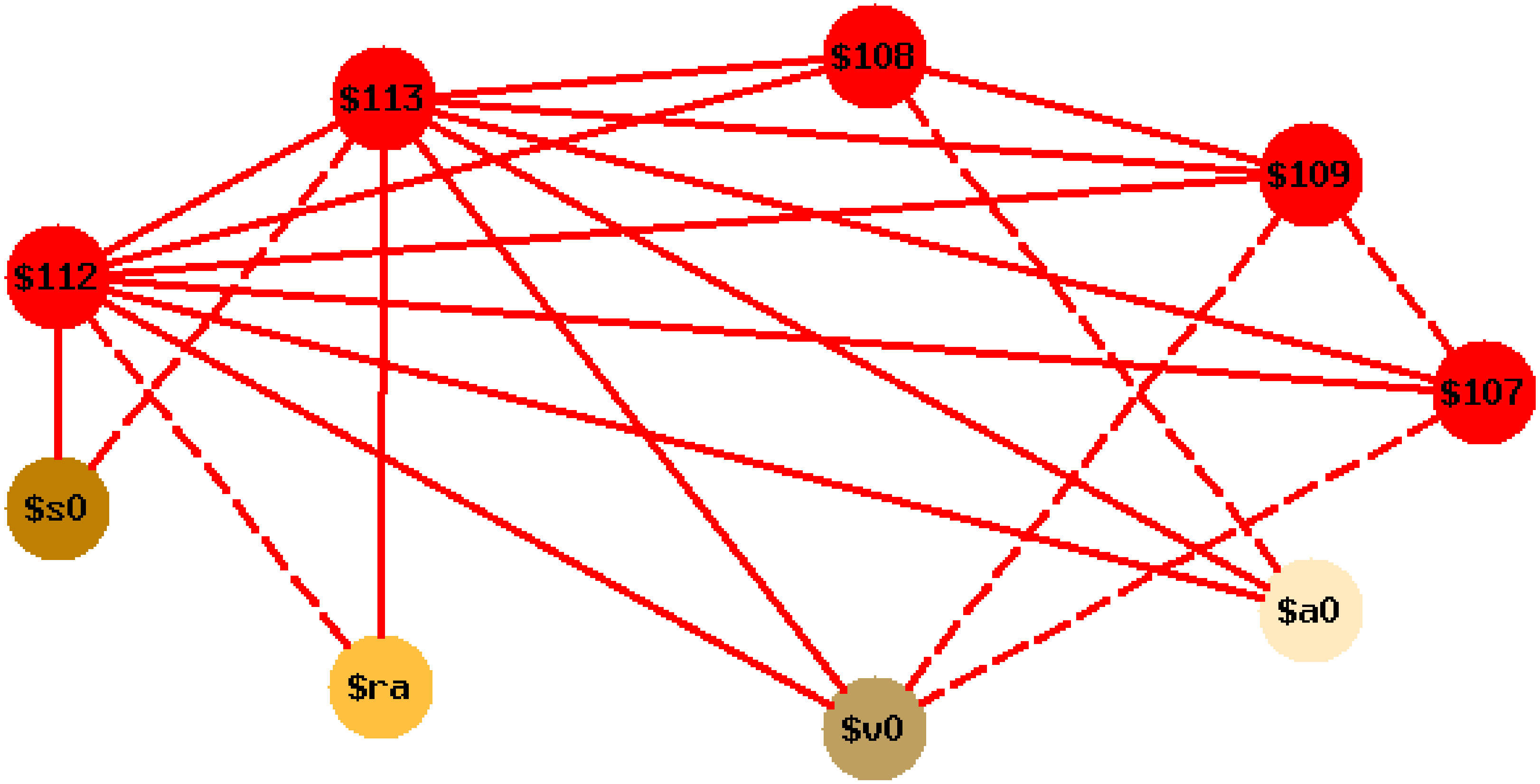
val spill_fun : Ass.temp Smallset.set -> Spim.procedure -> Spim.procedure

lw e_1 , 4(**$sp**) *# load* t_2
add e_2, e_1, 2
sw e_2 , 0(**$sp**) *# store* t_1
lw e_3 , 0(**$sp**) *# load* t_1
lw e_4 , 4(**$sp**) *# load* t_2
mul e_5, e_3, e_4
sw e_5 , 0(**$sp**) *# store* t_1

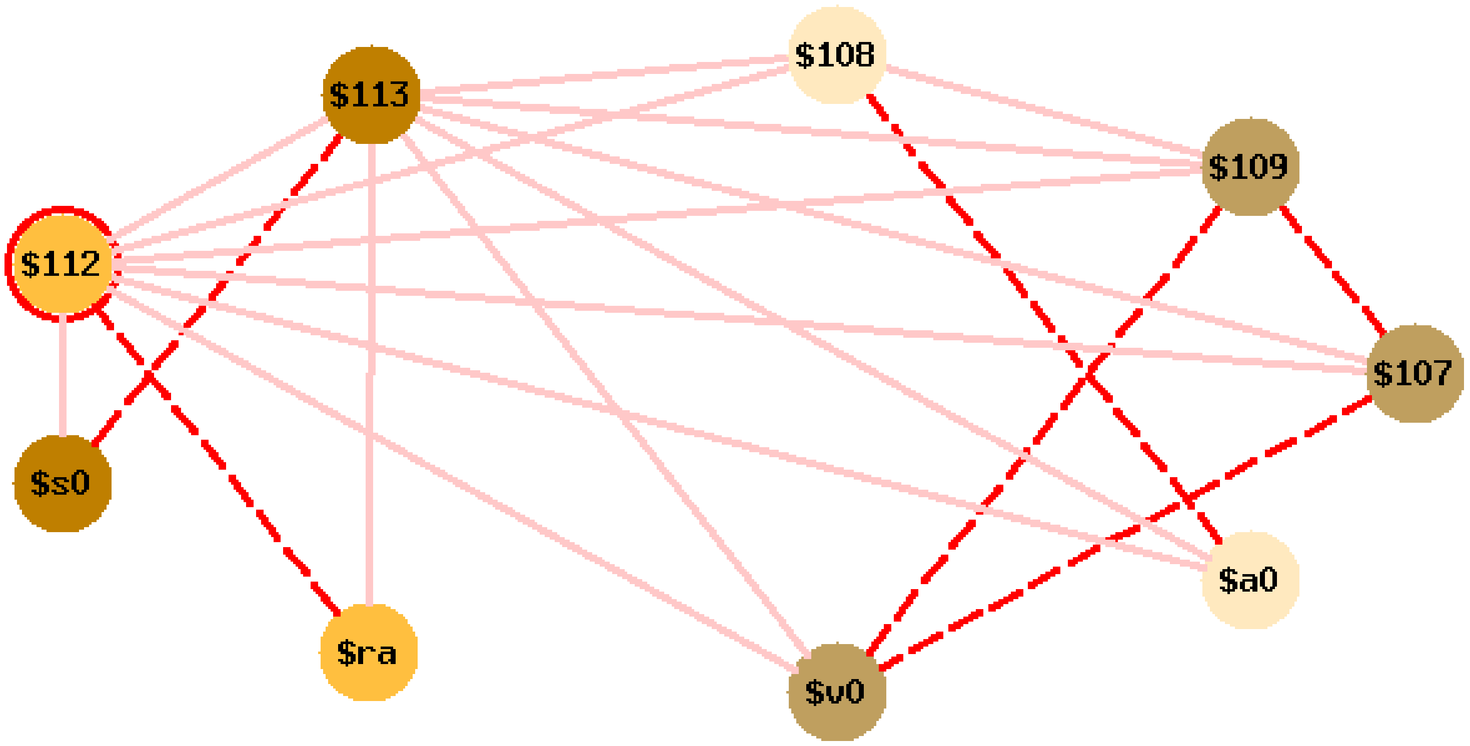
Les temporaires frais alloués comme *auxiliaires de spill* sont de durée de vie brève : ce sont des *éphémères*.

Coloriage de graphe

Attribuer des registres aux temporaires revient à attribuer des couleurs aux sommets du graphe d'interférence.



Graphe colorié



NB. Deux voisins (arcs d'interférence) portent des couleurs différentes.

Problème du coloriage de graphe

Soit un graphe G et K un nombre de couleurs.

Colorier G avec les K couleurs est NP-complet.

Degrés des sommets

faible Strictement moins de K voisins.

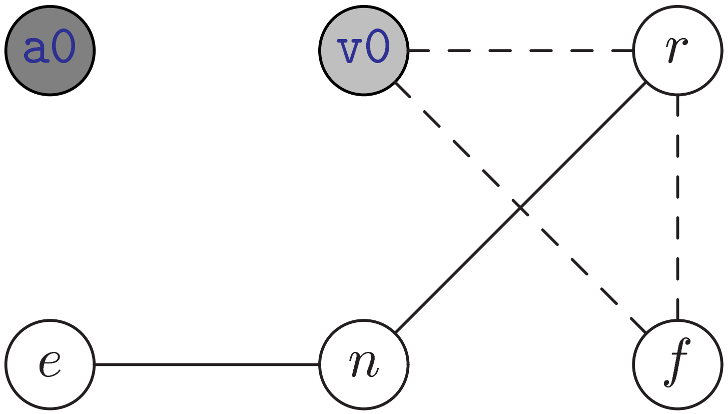
fort K voisins ou plus.

Remarque Si s est un sommet de faible degré et que le graphe $G \setminus \{s\}$ est K-coloriable, alors G est K coloriable.

Procédure récursive de coloriage

1. À la *descente*. Retirer les sommets de faible degré. Cela diminue le degré des sommets restants et permet de continuer au mieux jusqu'à ce que le graphe soit vide.
2. À la *remontée*. Dans ce cas le graphe est coloriable, et on est certain de pouvoir attribuer correctement les couleurs au retour de la procédure récursive.
3. Sinon, le graphe *peut* ne pas être coloriable.

Exemple



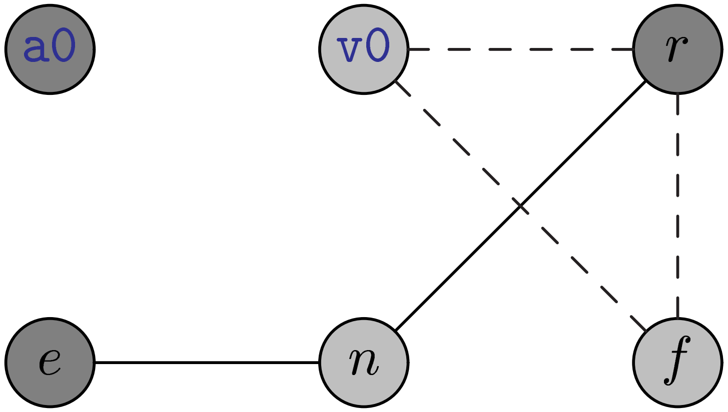
État initial :

- $K = 2$ (couleurs `a0` et `v0`).
- Faible degré : e r et f.
- Fort degré : n.

Une trace de l'algorithme simple

`low`	`high`	`removed`
e, f, r	n	
f, n, r		e
n, r		e, f
r		e, f, n
		e, f, n, r

`sommet`	`interdit`	`possible`	`choisie`
r		`a0`, `v0`	`a0`
n	`a0`	`v0`	`v0`
f		`a0`, `v0`	`v0`
e	`v0`	`a0`	`a0`



Réalisation

Plutôt que d'enlever les sommets (et les arcs) on garde le degré courant dans les sommets du graphe :

```
(* Contenu des noeud du graphe d'interférence *)
type interference = {
    temp : temp  ;                    (* un temporaire *)
(* Les champs suivants sont utiles pour l'allocation de registres *)
    mutable color : temp option ;
    mutable degree : int ;
    mutable elem : ((interference Sgraph.node) Partition.elem) option ;
    mutable occurs : int ;
  }
```

Et on gère une partition des sommets.

- À la descente : `precolored`, `low`, `high` et `removed`.
- À la remontée, rien pour le moment.

Interface des partitions (impératives)

```
type 'a t      (* type des sous-ensembles de la partition *)
type 'a elem (* type des éléments *)
val make : int -> 'a t array
        (* make n créer un vecteur de partitions *)
val create : 'a t -> 'a -> 'a elem
      (* « create s e » créer un élément e dans le sous-ensemble s *)
val info : 'a elem -> 'a
      (* « info e » retourne les informations sur l'élément e *)
val belong : 'a elem -> 'a t -> bool
    (* test d'appartenance *)
val move : 'a elem -> 'a t -> unit
    (* changement de sous-ensemble *)
val pick : 'a t -> 'a elem option
    (* « pick s » renvoie un élément de s, None si s est vide *)
```

Création des partitions

```
let sets = Partition.make 4

let precolored = sets.(0)
and low = sets.(1)
and high = sets.(2)
and removed = sets.(3)

module Mach = Spim
(* Mach.registers est la liste des registres machines *)
let colors = Smallset.of_list Mach.registers
and ncolors = List.length Mach.registers
```

Partitionement initial

On suppose que les champs de l'information `i` sont correctement initialisés.

```
let build_partition ig =
  Sgraph.iter ig
   (fun n ->
     let i = Sgraph.info ig n in
     let e = match i.color with
     | Some r -> Partition.create precolored n
     | None   ->
          if i.degree < ncolors then
            Partition.create low n
          else
            Partition.create high n in
     i.elem <- Some e)
```

Algorithme

```
let rec colorize ig = match Partition.pick low with
| Some e -> (* prendre un sommet de faible degré *)
    remove ig e   ;                  (* « l'enlever » *)
    if colorize ig then begin        (* colorier le reste du graphe *)
      let c = choose_color ig e in   (* colorier le sommet enlevé *)
      put_color ig e c ;
      true
    end else false
| None -> (* low est vide *)
   match Partition.pick high with
   | Some _ -> false
   | None   -> true
```

- Que doit faire la fonction `remove` ?
- Que doit faire la fonction `choose_color` ?

Que faire en cas d'échec ?

Il faut spiller un certain nombre de temporaires et recommencer.

On a intérêt à spiller :

- Un temporaire de degré conséquent.
- Un temporaire peu utilisé dans le code.

Quelques pistes :

- Une envie : combiner tentative de coloriage et choix des spills.
- Une question : que se passe-t-il si on « enlève » un sommet de fort degré.

Coloriage optimiste

Quand il n'y a plus de sommets de faible degré, enlever un sommet s de fort degré. À la remontée :

- Les $v \geq K$ voisins de s portent au total strictement moins de K couleurs distinctes, parce que ces voisins ont des couleurs identiques en nombre suffisant. On peut donc colorier s.
- Ou bien, il n'est pas possible de colorier s et on décide de spiller s.

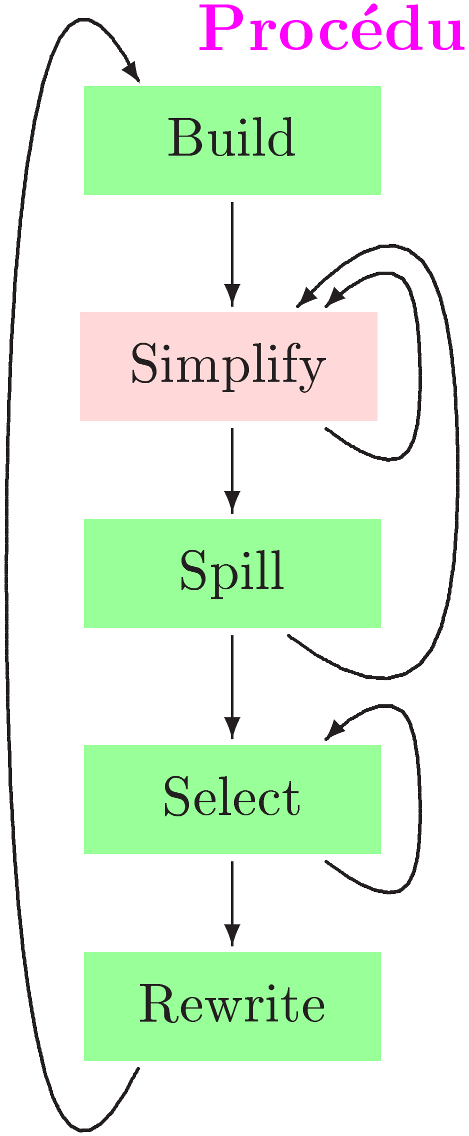
Le sommet s est un spill *potentiel*. Il faut bien le choisir.

Algorithme

Deux sous-ensembles supplémentaires `colored` et `removed`.

```
let rec colorize ig = match Partition.pick low with
| Some e -> (* prendre un sommet de faible degré *)
    remove ig e    ;                    (* « l'enlever » *)
    colorize ig    ;                    (* colorier le reste du graphe *)
    begin choose_color ig e with        (* choisir une couleur *)
    | Some c -> (* colorier *)
        put_color ig e (Some c) ; Partition.move e colored
    | None   -> (* spiller *)
        Partition.move e spilled
    end
| None -> (* low est vide *)
   match select_spill ig with (* selectionner un spill potentiel *)
   | Some e -> Partition.move e low ; colorize ig (* « l'enlever », continuer *)
   | None   ->    ()                                (* graphe vide, c'est fini *)
```

Procédure itérative (coloriage optimiste)



– Construire le graphe d'interférence

– Retirer un nœud de faible degré du graphe et l'empiler

– Retirer un nœud de fort degré du graphe et l'empiler

– Dépiler et colorier un nœud ou décider de le placer en pile

– Réécrire le code

Itération

Après une tentative de coloriage :

- Si `spilled` est vide. Effectivement colorier le code.
- Sinon, effectivement spiller et tout recommencer.

NB. Le spill introduit de nouveaux temporaires dans le code (les auxiliaires de spill). Mais ce sont des éphémères, qui interfèrent moins que les temporaires spillés.

Le graphe d'interférence du nouveau code est *plus simple* (penser par exemple au spill d'une sauvegarde de callee-save).

Terminaison Elle se produit en deux (plus rarement trois) tentatives :

- Si il y assez de registres (au moins trois).
- Si on éviter de spiller un auxiliaire de spill.

Une heuristique de spill possible

```
let cost ig e =
  let n = Partition.info e in
  let i = Sgraph.info ig n in
  (float i.occurs) /. (float i.degree) +.
  (if Gen.is_ephemere i.temp then 100.0 else 0.0)

let select_spill ig = Partition.pick_lowest (cost ig) high
```

Bon choix des couleurs

On utilise le coloriage *biaisé* (par les arcs `move`).

Couleur *possible* Une couleur qui n'est pas celle d'un voisin selon les arcs d'interférence.

Couleur *désirable* La couleur d'un voisin selon les arcs `move`.

Couleur *indésirable* La couleur désirable d'un voisin selon les arcs d'interférence.

À la remontée (étape *Select* de l'algorihme itératif). Préférer, dans l'ordre.

1. Une couleur désirable et possible.
2. Une couleur possible et non-indésirable.
3. Une couleur simplement possible.