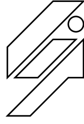


Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

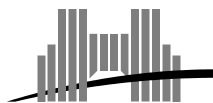
Register allocation and spill complexity under SSA

Florent Bouchez ,
Alain Darte ,
Christophe Guillon ,
Fabrice Rastello

August 2005

Research Report N° 2005-33

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Register allocation and spill complexity under SSA

Florent Bouchez , Alain Darte , Christophe Guillon , Fabrice Rastello

August 2005

Abstract

This report deals with the problem of choosing which variables to spill during the register allocation phase. Spilling is used when the number of variables is higher than the number of registers, and consists of storing the value of a variable in memory and loading it when necessary. The problem is that instructions dealing with memory are time-consumming. Hence the goal is to minimize the amount of spilled variables, which is a highly studied problem for compiler design, but nevertheless NP-complete. Meanwhile, a program under SSA form has the interesting property that on cases where spill is unnecessary, the problem of register allocation is not anymore NP-complete but polynomial. The interesting question is: can we solve the spilling problem under SSA, come back from SSA by splitting live-ranges as SSA does and finaly use classical register allocator?

We show in this report that unfortunately many formulations of the spilling problem are also NP-complete under SSA form. In particular, the node-deletion approach used in most compilers remains NP-complete on most cases even on basic-blocks. The only polynomial solution has a too high complexity in practice. But this first advanced study on the complexity of the spill problem under SSA greatly helps to the understanding and gives directions for polynomial approximations.

In this report, we also talk about the problem of splitting variables aggressively as in SSA. This shows the weakness of the "iterated register coalescing" regarding false interferences created by the adding of move instructions. We then implemented in a production compiler for st200 an interference graph based on liveness analysis and value numbering: two variables interfere if at one's definition the other is live and carries another value. The experiments we made and present in this report show that with such an interference graph aggressive splitting is not a problem.

Keywords: Compilation, register allocation, spill, coalescing, SSA, perfect graph, NP-completeness

Résumé

Les problèmes de l'allocation de registres et du vidage en mémoire (*spill*) sont nés avec la compilation ; ils ont été très étudiés mais sont NP-complets dans le cas général. Cependant, un programme sous forme SSA a la particularité de posséder un graphe d'interférences triangulé ce qui rend polynomiale la phase d'allocation de registres. Nous montrons dans ce rapport que malheureusement le problème du vidage en mémoire sous forme SSA reste NP-complet dans le cas général et même si l'on se restreint au bloc de base presque tous les cas sont NP-complets. L'algorithme polynomial trouvé dans un cas particulier est inutilisable car trop coûteux.
On trouvera donc dans ce rapport la première étude poussée de complexité du problème de vidage en mémoire sous forme SSA. Cette étude aide à la compréhension du problème et mène à des pistes pour des heuristiques polynomiales.
Ce rapport présente également un algorithme simple pour réduire le nombre d'instructions de copie créées lors du passage à la forme SSA. Il se base sur une étude des valeurs que contiennent les variables pour détecter les copies inutiles et allouer les variables correspondantes au même registre. Cet algorithme a été implanté avec succés dans un compilateur industriel utilisé par STMicroelectronics.

Mots-clés: Compilation, allocation de registres, vidage en mémoire, forme SSA, NP-complétude, réduction des copies, graphe parfait

Contents

1 Introduction 1
1.1 L'allocation de registres . . . 1
1.2 État de l'art . . . 1
2 Vidage des registres en mémoire 3
2.1 Définitions . . . 4
2.2 Coloriage . . . 6
3 Approche théorique 7
3.1 Problème général . . . 8
3.1.1 Intersection de sous-graphes . . . 8
3.1.2 Graphe d'intervalles . . . 10
3.2 Intervalles troués . . . 11
3.2.1 Nombre de trous non imposé . . . 13
3.2.2 Diminuer le nombre de variables vivantes d'une constante . . . 14
3.2.3 Diminuer jusqu'à une variable . . . 15
3.3 Conclusion . . . 19
4 Implantation 19
4.1 Motivations . . . 20
4.2 Analyse des valeurs . . . 20
4.2.1 Aspect théorique . . . 21
4.2.2 Aspect pratique . . . 23
4.2.3 Résultats . . . 25
4.3 Améliorations . . . 25
5 Conclusion 27

1 Introduction

1.1 L'allocation de registres

Dans ce rapport, nous nous intéresserons à un problème fondamental de la compilation : l'allocation de registres. Lors de la compilation d'un programme, il faut déterminer dans quels registres seront placées les variables nécessaires à l'exécution du programme. Ce problème est étroitement lié au problème du *spill* : le vidage des variables en mémoire. En effet, il arrive souvent que le nombre de variables existantes en un point du programme dépasse le nombre de registres disponibles ; dans ce cas il est nécessaire de sauvegarder certaines variables en mémoire pour diminuer le nombre de registres nécessaires.

On appelle communément la phase d'allocation de registres la phase de coloriage ; si l'on considère le nombre de registres disponibles comme un ensemble de couleurs, il faut en effet colorier les variables. On distingue deux principes généraux pour l'allocation de registres :

i. soit colorier les variables et faire du vidage quand il n'y a plus de couleur disponible ;

ii. soit déterminer à l'avance les endroits où le vidage sera nécessaire, vider certaines variables en mémoire, puis colorier le reste.

Pour ces deux méthodes, nous avons besoin de la notion d'*interférence* entre deux variables : deux variables interfèrent si elles sont vivantes en même temps en au moins un point du programme. Une variable vivante est une variable dont on ne doit pas perdre la valeur car elle sera utilisée plus tard. Cela signifie que deux variables qui interfèrent ne peuvent pas être assignées au même registre. Cette notion nous amène à la notion de *graphe d'interférence*, graphe dans lequel les sommets sont les variables et les arêtes les interférences. Nous sommes alors en présence d'un problème de coloration de graphe puisque deux sommets adjacents ne peuvent être coloriés avec le même registre.

Or le problème de la coloration de graphe est NP-complet si le graphe est quelconque ; et il est facile de construire à partir d'un graphe quelconque G un programme dont le graphe d'interférence est G : le problème de l'allocation de registre est ainsi NP-complet (preuve par Chaitin [6]) et on utilise donc en pratique des algorithmes non optimaux.

1.2 État de l'art

On retrouve plus souvent la méthode (i) dans la littérature contre quelques (vieux) articles pour la méthode (ii) comme [7] en 87 ou [11] en 92. Les algorithmes d'allocation de registres les plus utilisés sont des extensions ou améliorations de celui publié par Chaitin dans son article de 1982 *Register Allocation & Spilling via Graph Coloring* [5] ; c'est l'algorithme 1 de ce rapport et le principe est le suivant : on cherche à k-colorier un graphe donc un sommet qui a strictement moins de k voisins est colorable puisqu'il suffit de lui donner une couleur différente de ses voisins : on l'enlève du graphe et on continue jusqu'à ce que le graphe soit vide ou que tous les sommets aient un degré supérieur ou égal à k ; dans ce dernier cas on en enlève un qui devra peut-être être vidé en mémoire et on continue.

```
Données : k = nombre de registres
construire G le graphe d'interférence des variables ;
pile ← ∅ ;
tant que G ≠ ∅ faire
    tant que il existe n tel que deg(n) < k faire
1       empiler n ;
        retirer n de G ;
    fin
    si G ≠ ∅ alors
        soit n ∈ G ;
        /* n est un spill potentiel */
        empiler n ;
        retirer n de G ;
    fin
fin
2 pour chaque n dans la pile faire
    dépiler n ;
    si ∃ c couleur non prise par un voisin de n alors
        assigner n à c ;
        rajouter n à G ;
    sinon
        ajouter n à la liste des spill;
    fin
fin
```

Algorithme 1: Allocation de registres par coloration de graphe.

Explication détaillée : pendant la phase de dépilement (boucle 2), les sommets qui avaient été empilés par l'instruction ligne 1 sont tous coloriables puisqu'ils ont moins de k voisins. Les autres sont peut-être coloriables si plusieurs de leurs voisins ont la même couleur. Si des sommets n'ont pas été coloriés à la fin de l'algorithme, ils doivent être vidés en mémoire : on modifie le programme pour qu'ils soient stockés en mémoire après chaque définition (instructions *store*) et chargés de la mémoire dans une variable temporaire avant chaque utilisation (instructions *load*). Après cette phase, de nouvelles variables ont été créées : il faut recommencer le processus de coloriage et éventuellement de vidage. On itère ce processus jusqu'à ce que plus aucun vidage ne soit nécessaire ; il est prouvé que cette méthode converge, et en pratique il est rare d'effectuer plus de deux ou trois phases de coloriage.

Un inconvénient de la méthode de Chaitin est le *spill everywhere*: une variable vidée en mémoire est vidée *partout*. Il arrive qu'on ne manque de registres qu'à un endroit particulier, auquel cas une variable n'a besoin d'être vidée qu'à cet endroit ; elle pourrait être assignée normalement à un registre partout ailleurs. Ce n'est malheureusement pas évident à faire puisque la génération du code dû au vidage est faite durant la phase de coloriage et non pas après coup, ce qui empêche de faire des optimisations de code locales.

Il existe des méthodes de raffinement de l'algorithme de Chaitin pour tenter de remédier à ce problème : Briggs dans [4] montre qu'il est par exemple inutile de charger une variable vidée

si le précédent chargement est assez proche ; dans *Spill Code Minimization via Interference Region Spilling* [3], les auteurs présentent une méthode pour vider les variables uniquement sur les régions de code où elles interfèrent ; dans *Load/store range analysis for global register allocation*[13], les auteurs divisent la région où une variable est en vie en plusieurs régions au niveau des définitions et utilisations, le vidage d'une variable pourra alors être restreint à une de ces petites régions plutôt que d'être effectué sur tout le domaine d'existence.

Ces deux exemples introduisent une notion de *granularité*. L'idée commune est celle de *diviser* les variables (*variable splitting*) : une variable a s'appellera a_1 dans une partie du prorgamme, a_2 dans une autre, etc. Il sera alors possible de ne vider que a_2 si l'on arrive à colorier les autres a_i. Le problème de ces méthodes est de savoir où diviser une variable car chaque division crée une nouvelle instruction de copie (par exemple $a_2 \leftarrow a_1$) ce qui diminue l'efficacité du code.

Plan du rapport : la division à laquelle nous nous sommes intéressés est celle induite par une représentation du programme sous forme SSA, forme connue depuis longtemps ; elle est particulièrement intéressante pour le coloriage et des résultats existants (que nous présenterons) annonçaient des possibilités d'algorithmes polynomiaux. Dans la partie 2 nous introduirons les notations utilisées dans ce rapport et présenterons des propriétés pour la plupart bien connues (mais certaines sont nouvelles) qui forment la base de l'allocation de registres, du vidage et de la forme SSA. Puis la partie 3 expliquera notre tentative de percée du problème de vidage en utilisant les propriétés de la forme SSA ; nous y présentons de nouveaux résultats de NP-complétude ainsi qu'un nouveau résultat de polynomialité. Il en résulte une vision globale de la complexité du problème de vidage qui n'existait pas auparavant. Enfin la partie 4 concerne une implantation d'un algorithme dans un compilateur de production. La raison de cette implantation est que la forme SSA crée de fausses interférences non détectées par la structure de donnée de Chaitin aussi nous avons formalisé une nouvelle définition de l'interférence. L'implantation combine cette dernière avec un algorithme d'analyse de flot de données (classique du domaine de la compilation) pour améliorer l'algorithme d'allocation de registres.

2 Vidage des registres en mémoire

Pour commencer à travailler sur le problème du vidage, il nous a semblé nécessaire de commencer par re-définir précisément le cadre de travail, d'énoncer les propriétés de base et de les démontrer, cette étape étant souvent oubliée dans la littérature. Nous verrons également que sous SSA, le problème du coloriage est polynomial car on travaille sur un graphe triangulé. Ce résultat est nouveau car il n'est utile qu'avec l'approche (ii) (vidage d'abord, allocation ensuite) qui a été très peu abordée. Pour utiliser cette propriété, il nous faudra d'abord résoudre le problème du vidage en mémoire ce qui revient finalement à faire du coloriage de graphe dans lequel on s'autorise à supprimer (vider) certains sommets : c'est un *node deletion problem* mais pour un sous cas (celui des graphes triangulés) que Yannakakis [14] ne traite pas (problème [GT21] de Garey et Johnson [9] : *Induced Subgraph with Property* Π). Nous traiterons ce cas dans la partie 3.

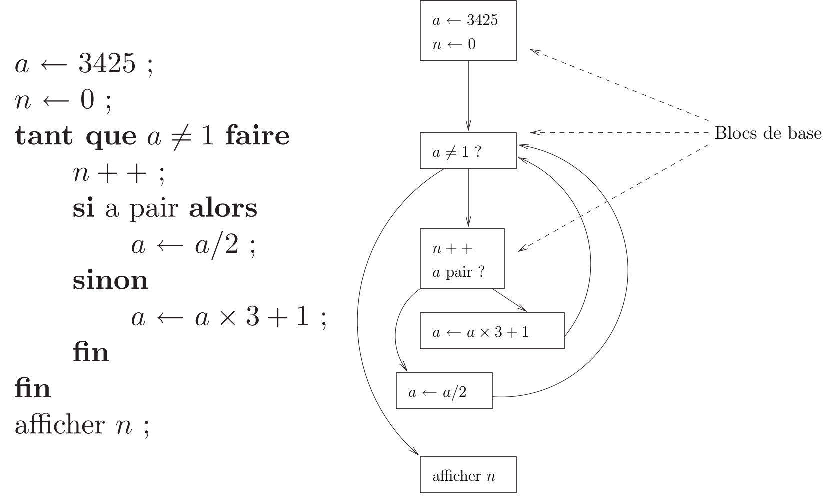


Figure 1: Graphe de contrôle de flot d'un programme

2.1 Définitions

Définition. Un **bloc de base** (*basic bloc*) est une suite maximale d'instructions sans branchement : il n'y a pas d'autre chemin possible au milieu d'un bloc de base. Un programme peut être représenté par un **graphe de contrôle de flot**, graphe dans lequel les sommets sont des blocs de base et les arêtes orientées des chemins possibles lors de l'exécution du programme. La figure 1 montre un exemple de programme et son graphe de contrôle de flot correspondant.

Définition. Une instruction s (ou un bloc d'instructions) **domine** une autre instruction t si et seulement si tout chemin sans cycle (i.e. qui contient au plus une fois chaque instruction) de la racine du programme à t passe par s. On note alors $s \succ t$.

Théorème 1. *La dominance est une relation d'ordre partiel : elle est antisymétrique, reflexive et transitive.*

Proof.
Prouvons les trois propriétés qui caractérisent un ordre partiel :

- antisymétrie : si $s \succ t$ et $t \succ s$. Soit un plus court chemin de ràt. $s \succ t$ donc il passe par s : $r \rightsquigarrow^1 s \rightsquigarrow^2 t$, alors comme t domine s, le chemin de la racine à s passe par t donc $r \rightsquigarrow^* t \rightsquigarrow^{1'} s \rightsquigarrow^2 t$. Comme on a pris le plus court chemin, $r \rightsquigarrow^* t$ est de longeur supérieure ou égale à $r \rightsquigarrow^1 s \rightsquigarrow^2 t$ ce qui n'est possible que si $t \rightsquigarrow^{1'} s$ et $s \rightsquigarrow^2 t$ sont de longueur nulle : c'est-à-dire que $s = t$.

- reflexive : s est le dernier élément de tout chemin de la racine à lui-même, donc il se domine ;

- transitive : si $s \succ t$ et $t \succ u$, alors soit un chemin $r \rightsquigarrow u$ de la racine à u. Il contient t donc il se décompose en $r \rightsquigarrow t \rightsquigarrow u$. Or $s \succ t$ donc on peut décomposer $r \rightsquigarrow t$ en $r \rightsquigarrow s \rightsquigarrow t$ donc $r \rightsquigarrow u$ contient s et donc $s \succ u$.

■

Définition. Forme SSA : *Static Single Assignement* ; chaque variable est définie textuellement[1] une seule fois avant d'être utilisée. Pour une variable a, on note $\mathbf{def}(a)$ l'instruction où a est définie et $\mathbf{use}(a)$ l'*ensemble* des instructions qui utilisent a. La condition suivante doit être respectée : $\mathrm{def}(a)$ domine tous les éléments de $\mathrm{use}(a)$.

En SSA, il y a donc une unique définition *statique*, mais il peut y avoir plusieurs définitions *dynamiques*, par exemple si la définition a lieu dans une boucle.

Définition. Le **graphe de dominance** est le graphe de Hasse[2] du graphe où les sommets sont les instructions et les arêtes (orientées) représentent les relations de dominance : $s \to t \Leftrightarrow s \succ t$.

Propriété 1. Si s et t dominent u, alors soit s domine t, soit t domine s.

Proof. Soit un chemin (sans cycle) de la racine r à u. Ce chemin contient s et t par définition. On peut supposer par symétrie que s est avant t sur ce chemin : $r \rightsquigarrow s \rightsquigarrow t \rightsquigarrow^+ u$. S'il existait un chemin $r \rightsquigarrow^* t$ allant de la racine à t ne passant pas par s, alors la prolongation de ce chemin jusqu'à u, $r \rightsquigarrow^* t \rightsquigarrow^+ u$, serait un chemin de la racine à u qui ne passe pas par s ce qui contredit le fait que $s \succ u$. Donc tous les chemins de r à t passent par s donc $s \succ t$. ■

Théorème 2. *Le graphe de dominance sous SSA est un arbre.*

Proof. Un nœud u ne peut avoir qu'un seul prédécesseur direct : si s et t dominent u, alors d'après la propriété 1 l'un des deux domine l'autre, par exemple $s \succ t$ et donc $s \to u$ est une arête transitive donc elle n'apparaît pas dans le graphe de Hasse. De plus le graphe est bien connexe puisque la racine domine tous les autres nœuds. ■

Définition. Le **domaine de vie** d'une variable a, noté $\mathrm{vie}(a)$, est l'ensemble des instructions comprises entre $\mathrm{def}(a)$ et les $\mathrm{use}(a)$. C'est un sous-graphe du graphe de flot et un sous-arbre du graphe de dominance sous SSA.

Définition. Deux variables $a \neq b$ **interfèrent** si et seulement si elles sont vivantes en même temps, c'est-à-dire que leurs domaine de vie s'intersectent.

Propriété 2. Sous SSA, si a interfère avec b alors soit $\mathrm{def}(a) \succ \mathrm{def}(b)$ soit $\mathrm{def}(b) \succ \mathrm{def}(a)$.

Proof. Soit s un point où a et b sont vivantes en même temps. Alors $\mathrm{def}(a) \succ s$ et $\mathrm{def}(b) \succ s$. La propriété 1 permet de conclure. ■

Corollaire 1. *Sous SSA, si a interfère avec b avec* $\mathrm{def}(a) \succ \mathrm{def}(b)$*, alors* $\mathrm{def}(b) \in \mathrm{vie}(a)$.

Proof. $\mathrm{def}(a) \succ \mathrm{def}(b)$ or a et b interfèrent donc a est vivante en $\mathrm{def}(b)$ et donc $\mathrm{def}(b) \in \mathrm{vie}(a)$. ■

Définition. Un graphe non orienté est **triangulé** si tout cycle de taille supérieure ou égale à quatre possède une corde (arête entre deux sommets non adjacents du cycle).

[1] dans le code source du programme

[2] graphe où les arêtes transitives sont supprimées

Le théorème suivant n'apparaît pas dans la littérature. Jusqu'à présent personne ne l'avait remarqué mais on peut trouver dans [7] des idées qui y aurait mené. C'est ce théorème, à cause du caractère parfait des graphes triangulés, qui nous a poussés à chercher dans la direction du vidage sous SSA.

Théorème 3. *Un graphe d'interférences sous SSA est triangulé.*[3]

Proof. Soit un graphe d'interférences sous SSA. On définit une orientation des arêtes selon la dominance : si $\mathrm{def}(a) \succ \mathrm{def}(b)$, alors $a \rightarrow b$. La propriété 2 assure que toute arête est orientée.

(F1) Dans ce graphe orienté, il n'existe pas de chemin cyclique car la dominance est anti-symétrique : s'il existait un chemin cyclique, alors tous les sommets du chemin seraient égaux ;

(F2) soit un cycle non orienté de taille supérieure ou égale à quatre du graphe s'il en existe, alors d'après (F1) il existe a dans ce cycle dont les voisins dans le cycle sont b et c tel que $b \rightarrow a \leftarrow c$;

(F3) si l'on applique le corollaire 1 à (F2), $\mathrm{def}(a) \in \mathrm{vie}(b)$ et $\mathrm{def}(a) \in \mathrm{vie}(c)$, donc $\mathrm{vie}(b) \cap \mathrm{vie}(c) \neq \emptyset$. Ainsi b et c interfèrent ce qui signifie qu'il existe une arête entre b et c dans le graphe. Donc le cycle possède une corde : le graphe est bien triangulé.

■

2.2 Coloriage

Un des avantages de la forme SSA est celui du coloriage.
La propriété suivante permet de faire le lien entre le nombre de variables en vie dans le programme et les cliques du graphe d'interférences dont la taille détermine le nombre de couleur nécessaires.

Propriété 3. Dans un graphe d'interférence sous SSA :

- À chaque clique de G de taille Ω correspond un point du programme où Ω variables sont en vie.
- Réciproquement, pour tout point du programme, l'ensemble des variables en vie en ce point forme une clique de G.

Proof. Le deuxième point est évident par définition de G. Démonstration du premier point : si deux variables interfèrent alors la définition de l'une domine celle de l'autre d'après la propriété 2, ce qui définit un ordre total sur une clique. Soit a le plus petit élément. $\mathrm{def}(a)$ est contenue par tous les domaines de vie des autres variables. Donc au point $\mathrm{def}(a)$ au moins, toutes les variables de la clique sont en vie. ■

[3]Puisque le domaine de vie sous SSA est un sous-arbre du graphe de dominance, il suffirait d'appliquer directement un théorème de Golumbic [10]. Cependant, c'est un théorème difficile et nous avons préféré faire une démonstration entière et simple.

Or la taille de la plus grand clique, k est le nombre chromatique du graphe : celui-ci est k-colorable. Le théorème suivant permet de conclure :

Théorème 4. *Soit G un graphe triangulé k-colorable. Alors G est k-glouton-colorable, c'est-à-dire que l'algorithme 1 (de Chaitin) réussit à k-colorier le graphe sans vider de variable en mémoire.*

Proof. Le point clé de la démonstration provient d'un théorème démontré par Golumbic dans [10] : tout graphe triangulé possède au moins un sommet simpliciel. Un sommet simpliciel est un sommet dont le voisinage forme une clique.

Soit s un sommet simpliciel de G et ν l'ensemble de ses voisins. Alors $\{s\} \cup \nu$ est une clique ; cette clique est de taille au plus k puisque G est k-colorable. Donc ν est une clique d'au plus $k-1$ sommets et s a strictement moins de k voisins. Ainsi l'algorithme de Chaitin peut supprimer ce sommet de G pour l'empiler. Comme un sous-graphe d'un graphe triangulé est triangulé (et aussi k-colorable), on peut recommencer avec G privé de s.

Ainsi, à tout moment de l'algorithme de Chaitin, il existe au moins un sommet de degré inférieur à k donc tous les sommets du graphe sont empilés en sachant qu'on pourra les colorier. La phase de dépilement réussit donc à assigner une couleur à tous les sommets et on obtient une k-coloration de G. ■

Sous SSA, on sait donc faire l'allocation de registres de manière optimale avec k registres si en tout point du programme au plus k variables sont en vie. Reste alors le problème du vidage : si l'on n'a que r registres, et que $k > r$ est la taille de la plus grande clique, on souhaite être capable de vider suffisamment de variables en mémoire pour passer à $k = r$ et pouvoir utiliser l'algorithme de Chaitin.

3 Approche théorique

Le problème du vidage sous SSA est NP-complet. C'est ce nouveau résultat que nous commencerons par exposer dans cette partie. Par contre, il existe plusieurs résultats connus de polynomialité sur les blocs de base dont nous ferons une compilation. On introduira alors une nouvelle notion : celle de trou . Elle permet de définir exactement le problème du vidage car il se trouve que la version habituellement considérée (celle utilisée pour la partie 3.1) est trop simple par rapport au problème réel. Nous exposerons alors les nouveaux résultats que nous avons obtenus avec notre notion de trou.

Énoncé du problème : on se donne un graphe d'interférences G ainsi qu'une fonction de poids sur les sommets. Cette fonction représente le coût pour vider la variable correspondante en mémoire (par exemple, elle peut être égale au nombre de stockages et de chargements à ajouter au code source pour la vider).
On appelle $\Omega = \omega(G) = \max_{c \in cliques(G)} \text{taille}(c)$, le nombre chromatique de G. C'est donc une donnée du problème et non une constante.

On suppose $\Omega > r$ le nombre de registres disponibles (donnée du problème). Le but est de diminuer Ω en vidant certaines variables en mémoire. Distinguons les trois problèmes suivants :

- $\Omega - 1$: passer de Ω à $\Omega - 1$;
- $\Omega - k$: passer de Ω à $\Omega - k$ avec k constante donnée ;
- $\Omega \searrow r$: passer de Ω à la variable r.

Remarquons la hiérarchie de difficulté des problèmes de $\Omega - 1$ à $\Omega \searrow r$: un algorithme polynomial pour un problème résoud les problèmes précédents ; et une preuve de NP-difficulté implique la difficulté des problèmes suivants.

3.1 Problème général

Dans cette partie, on s'intéresse au problème classique : vider des variables pour que, en tout point du programme, on ait au plus $\Omega - 1$, $\Omega - k$ ou r variables en vie. Le vidage est classique (pas de trous à l'inverse de la partie 3.2) : quand on vide une variable, le nombre de variables en vie diminue de 1 en tout point de son domaine de vie. Les résultats sont très différents selon que l'on se place dans le cadre d'un bloc de base ou d'un graphe de contrôle de flot général et sont exposés dans le tableau suivant (les problèmes étiquetés $\mathbb{P}$ sont polynomiaux tandis que ceux étiquetés $\mathbb{NP}$ sont NP-difficiles) :

	Bloc de base	Cas général
poids $= 1$	$\Omega \searrow r$ $\mathbb{P}$ glouton 3.1.2	$\Omega - 1$ $\mathbb{NP}$ (*3-exact cover* 3.1.1)
poids $\neq 1$	$\Omega \searrow r$ $\mathbb{P}$ ILP flow 3.1.2	$\mathbb{NP}$(cf ci-dessus)

3.1.1 Intersection de sous-graphes

Si l'on ne se restreint pas à un bloc de base, les domaines de vie des variables peuvent dépasser la portée d'un seul bloc et à une variable correspond donc un sous-graphe du graphe de contrôle de flot délimité par la définition et par les utilisations les plus extrêmes.

Nous allons voir que dans ce cas, même le problème $\Omega - 1$ avec poids unitaires est NP-complet. C'est un problème de type *node deletion* dans le cas particulier des graphes triangulés mais il n'est pas traité par Yannakakis dans son article [14] (problème [GT21] de Garey et Johnson [9] : *Induced Subgraph with Property* Π).

Problème : soit un graphe d'interférences sous SSA (donc triangulé). Ce graphe est Ω-colorable (taille de la plus grande clique). Trouver un ensemble de sommets de cardinal minimum tel que le graphe privé de ces sommets est $\Omega - 1$-colorable est NP-complet.

Proof. La réduction est faite du problème *3-exact cover* (problème [SP2] du Garey et Johnson [9]). Soit une instance de ce problème : E est un ensemble à $3n$ éléments $\{e_1, e_2, \cdots, e_{3n}\}$,

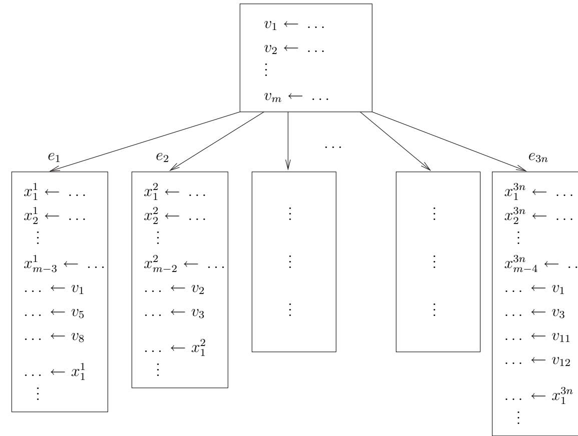


Figure 2: Réduction de *3-exact cover*

et $\mathcal{S} = \{v_1, v_2, \cdots, v_m\}$ un ensemble de sous-ensembles de E contenant chacun 3 éléments de E. On cherche $S \subset \mathcal{S}$ de cardinal n qui couvre E : tous les éléments de E appartiennent à un élément de S. Ce problème est NP-complet.

Construisons un programme définissant m variables $\{v_1, \cdots, v_m\}$ puis effectue un test qui mène à $3n$ branches possibles (voir figure 2). Dans chaque branche e_i on utilise toutes les variables v_j telles que $e_i \in v_j$ dans le problème initial. Avant leur utilisation, on définit des variables temporaires locales à chaque branche pour atteindre la pression registres m (nombre de variables simultanément en vie). Ces variables locales sont utilisées après l'utilisation des v_j pour qu'elles soient vivantes.

Dans ce cas, nous sommes bien sous forme SSA puisque les définitions sont uniques et $\Omega = m$ puisqu'il y a par endroits m variables simultanément en vie, notamment dans toutes les branches. Pour passer à $\Omega - 1$, il faut donc vider en mémoire au moins une variable existant dans chaque branche. On suppose qu'il y a au moins une variable de type v_j dans chaque branche (sinon, il existe un élément e_i qui ne peut pas être couvert et le problème n'a pas de solution).

Supposons que l'on a vidé toutes les variables d'un ensemble S. On suppose que S est une solution optimale : de cardinal minimum (poids unitaires) et en tout point du programme au plus $\Omega - 1$ variables sont en vie.

- Si $|S| = n$, alors on n'a pas pris de variable temporaire x_j^i car on a besoin de couvrir $3n$ branches et que seules les variables v_j couvrent plus d'une branche. Donc on n'a vidé que des variables correspondant aux ensembles v_j du problème initial et ils couvrent tous les éléments e : c'est une solution à *3-exact cover*.

- Si $|S| > n$, alors il n'y a pas de solution au problème de *3-exact cover*. En effet, supposons que l'on a une solution S' au problème *3-exact cover*, alors on vide toutes les variables correspondant aux éléments de S'. Comme S' couvre exactement tous les e_i, il y a une et une seule variable vidée par branche donc toutes les branches sont à $\Omega - 1$. Dans le bloc d'initialisation, il y a au moins une variable vidée donc il est au plus à $\Omega - 1$. Nous avons donc une solution à notre problème de vidage de coût n ce

qui contredit l'optimalité de S.

L'appartenance à NP étant évidente, on en déduit la NP-complétude du problème. ■

Nous avons donc une réponse partielle au *node deletion problem* de Yannakakis : même sur un graphe triangulé, ce problème reste NP-complet. Aussi nous allons nous pencher sur une version réduite du problème où l'on se restreint à un bloc de base.

3.1.2 Graphe d'intervalles

Le graphe d'interférences est un graphe d'intervalles si l'on est dans le cas d'un bloc de base : à chaque variable correspond un intervalle simple, qui commence à la définition de la variable (unique puisqu'on est sous SSA) et se termine à sa dernière utilisation. Cette simplicité du problème conduit à des algorithmes polynomiaux, mais malheureusement souvent oubliés dans la littérature actuelle Nous rappelons donc ces résultats à garder en mémoire.

Poids $= 1$ **:** c'est le cas où le coût de l'unique stockage (*store*) est égal à 1 et celui de chaque chargement (*load*) égal à 0. C'est polynomial : un simple algorithme glouton suffit. On rappelle que r est le nombre de registres disponibles.

1. trouver le premier point en partant du début du bloc où plus de r intervalles coexistent ;
2. supprimer un intervalle qui contient ce point et qui se termine le plus tard possible ;
3. recommencer en 1 jusqu'à ce que moins de r intervalles coexistent en tout point du bloc de base.

Coût des allocations en mémoire : nombre d'intervalles supprimés. Cet algorithme est optimal car il supprime un nombre minimal d'intervalles.

Proof. Soit $I_{glouton} = \{I^1_{glouton}, I^2_{glouton}, \cdots I^n_{glouton}\}$ la solution trouvée par l'algorithme glouton avec $I^j_{glouton}$ l'intervalle choisi par l'algorithme à l'étape j. Considèrons une solution optimale dont les intervalles sont $\mathcal{I}_{optimal}$ et telle que $I^1_{glouton}, \cdots I^i_{glouton} \in \mathcal{I}_{optimal}$ avec i maximal. Supposons que $i < n$. Soit p le point du programme où l'algorithme glouton a choisi $I^{i+1}_{glouton}$.

L'union des intervalles choisis par l'algorithme glouton recouvre bien le début du bloc de base jusqu'à p : pour tout point p' du bloc de base avant p s'il y a x variables en vie en trop, alors au moins x intervalles parmi $I^1_{glouton} \dots I^i_{glouton}$ contiennent p' d'après la construction de la solution gloutonne.

Au niveau du point p, l'algorithme glouton a choisi un intervalle, donc il y a au moins une variable en vie en trop. Donc il existe au moins un intervalle $I \in \mathcal{I}_{optimal}$ qui n'est pas dans $I_{glouton}$ et qui couvre le point p. Alors on échange I avec $I^{i+1}_{glouton}$ dans la solution optimale. L'ensemble d'intervalles obtenu a le même cardinal donc est optimal puisque les poids sont unitaires ; et c'est une solution :

- avant p les intervalles $I^1_{glouton} \cdots I^i_{glouton}$ couvrent le bloc de base,
- après p, $I^{i+1}_{glouton}$ va plus loin que I par construction de $I_{glouton}$, donc couvre au moins autant que I.

Donc il n'y a pas de défaut de couverture et la solution optimale modifiée reste une solution optimale.

Ceci est contradictoire avec la maximalité de i puisqu'on a trouvé une solution optimale qui contient les $i + 1$ premiers intervalles de $I_{glouton}$. Donc $i = n$ et l'algorithme glouton fournit une solution optimale.

■

Coût du stockage nul. Ce cas est différent de tous les autres cas mentionnés dans ce rapport en ce sens que l'on va pouvoir vider facilement une variable sur une *partie* de son intervalle de vie (à l'inverse des autres problème où l'on vide les variables sur *tout* le domaine de vie). Si le coût du stockage en mémoire (*store*) est nul : le problème est équivalent au problème de la gestion des pages de mémoire virtuelle (*page fault*) et est polynomial d'après Belady [2]. Le principe général est le suivant : comme le stockage ne coûte rien, on considère que toutes les variables sont disponibles en mémoire. On parcours le bloc de base à partir de la première instruction : dès qu'on doit vider une variable, on choisit celle dont la prochaine utilisation est la plus tardive (parmi les variables non vidées à cet endroit). Cette variable sera alors vidée entre la précédente utilisation et la suivante ce qui coûte exactement un chargement.

Cas général : poids quelconques. Si l'on cherche comme précédemment à vider les variables sur des portions de leur intervalle de vie, le problème est NP-complet d'après Farach et Liberatore [8].

Par contre si on se replace dans le cas où l'on vide les variables sur tout leur intervalle (*spill everywhere*), le problème est polynomial. On fait de la programmation linéaire en nombres entiers particulière : *ILP flow*. La résolution est effectuée dans $\mathbb{Q}$ ce qui est polynomial mais la particularité du problème (la matrice de l'hypergraphe d'incidence est unimodulaire) fait que les solutions obtenues sont toujours dans $\mathbb{Z}$. Le résultat n'a jamais été vraiment écrit mais si l'on étudie l'algorithme de 2-approximation de Farach et Liberatore [8] de la version longue de leur article *On Local Register Allocation*, on s'aperçoit que trouver une solution optimale dans notre cas se fait en temps polynomial.

3.2 Intervalles troués

Cependant les résultats précédents ne sont pas satisfaisants pour toute architecture de machine pour la raison suivante : lorsqu'on choisit de vider une variable, il faut (si l'architecture et de type RISC où les opérations se font uniquement entre registres) modifier le code source pour y insérer des instructions de stockage et de chargement, et il apparaît de nouvelles variables dont la durée de vie est très courte mais qui ont elles aussi besoin d'être assignées à un registre (comme dans l'algorithme 1 de Chaitin, qui doit alors recommencer l'allocation de

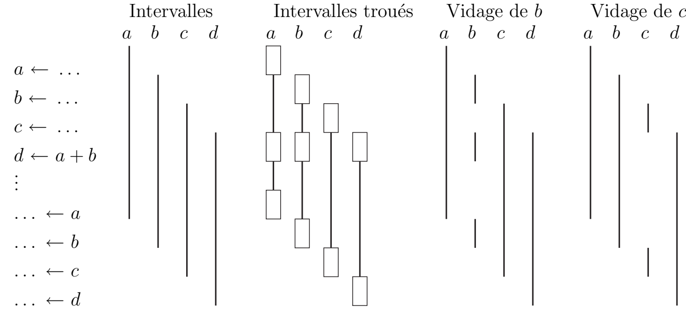


Figure 3: Exemple d'intervalles à trous

registres). Lorsqu'on vide une variable, on diminue donc de un le nombre de variables en vie en tout point du domaine de vie de cette variable, $s$auf au niveau des instructions où elle est définie ou utilisée. Le vidage d'une variable n'enlève donc pas l'intervalle correspondant à la variable mais une version trouée de cet intervalle, puisqu'il reste ponctuellement des petits intervalles.

Un exemple est présenté figure 3 ; supposons que l'on n'a que trois registres. Il faut donc vider une variable puisqu'après la définition de d, les quatre variables sont en vie. Si l'on considère les domaines de vie des variables comme de simples intervalles, alors on peut enlever n'importe laquelle des variables. Cependant on s'aperçoit que si on vide b, son utilisation lors de la définition de d nous oblige à charger b de la mémoire vers un registre ; le vidage de b est donc inutile. Par contre, vider la variable c résoud le problème.

Le problème est donc le suivant : enlever le minimum d'intervalles troués de manière à ce que le nombre maximum d'intervalles existant en tout point du programme (y compris les petits intervalles) passe de Ω à $\Omega - 1$, $\Omega - k$ ou r selon la version du problème.

On se restreint au cas du bloc de base ; nous verrons que même cette version simple du problème est difficile dans la plupart des cas. Dans le tableau récapitulatif suivant, on note t le nombre de trous, c'est-à-dire le nombre maximum d'intervalles qui peuvent être troués au même endroit du programme. Ce nombre correspond au nombre maximum de registres qui peuvent être utilisés dans une instruction. Par exemple pour une addition à trois arguments `add %reg1, %reg2 => %reg3`, on a $t = 3$.

	$t = 1$	$t = 2$	t non borné
poids $= 1$	?	$\Omega \searrow r$ $\mathbb{NP}$ (*independent-set* 3.2.3)	$\Omega - 1$ $\mathbb{NP}$ (*set-cover*, 3.2.1)
poids $\neq 1$	$\Omega \searrow r$ $\mathbb{NP}$ (3.2.3)	$\Omega - k$ $\mathbb{P}$ en $O(mn^{6(t+k)})$ 3.2.2	$\mathbb{NP}$ (cf ci-dessus)

On remarquera que dans le cas simplifié à l'extrême où un seul trou est autorisé et les poids sont unitaires, nous ne connaissons pas la difficulté du problème $\Omega \searrow r$. Cependant, ce cas est tellement éloigné des problèmes effectivement rencontrés en allocation de registres qu'un algorithme polynomial n'aurait aucune autre utilité pratique mais seulement théorique.

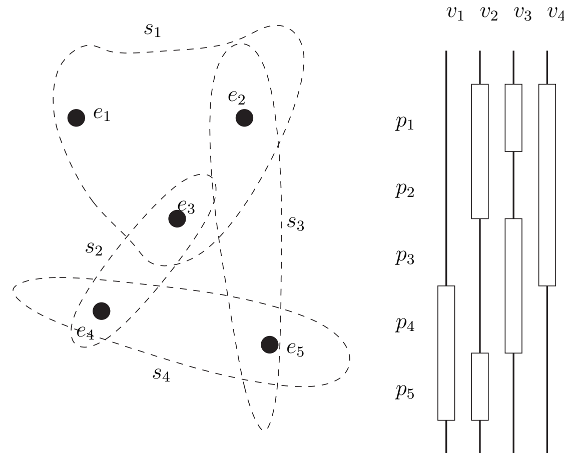


Figure 4: Exemple de couverture par ensemble.

3.2.1 Nombre de trous non imposé

Si le nombre de trous simultanément autorisés n'est pas fixé nous pouvons écrire des instructions qui utilisent autant de variables que l'on veut. Ce qui signifie que nous avons un contrôle total sur la forme des intervalles troués correspondant aux variables : sur tout un bloc de base on troue un intervalle aux instructions p que l'on souhaite en utilisant la variable associée dans p. Ceci nous a permis de faire très facilement une réduction de la couverture par ensembles (*set-cover*) ([SP5] du Garey et Johnson [9]).

Proof. Soit E un ensemble de n éléments et $\mathcal{S}$ un ensemble de m sous-ensembles de E. On cherche $S \subset \mathcal{S}$ de cardinal mimimum qui couvre E : pour tout e de E, il existe $s \in S$ tel que $e \in s$.

Créons alors un bloc de base de taille n dans lequel m variables sont vivantes en tout point. À l'instruction p_i, seules les variables v_j telles que $e_i \in s_j$ ne sont pas utilisées. Voir figure 4 : par exemple l'ensemble s_3 contient e_2 et e_5 donc la variable v_3 n'a *pas* de trou en p_2 ni en p_5, par contre elle est utilisée pour les trois autres instructions. On cherche à passer à $m - 1$ variables vivantes en tout point du bloc de base.

Soit une solution S pour la couverture par ensembles. Alors pour tout $s_j \in S$, on vide la variable v_j ce qui a pour effet de diminuer de 1 le nombre de variables en vie sur toutes les instructions p_i telles que $e_i \in s_j$ et de ne rien changer pour les autres instructions car la variable est utilisée donc elle devra être chargée de la mémoire. Puisque S est une couverture, cela signifie que pour toute instruction, au moins une variable non utilisée est vidée ce qui résoud notre problème de vidage.

Réciproquement, si l'on a une solution optimale pour le vidage en mémoire, alors pour tout point p_i, on a vidé au moins une des variables non utilisées. Soit v_j une de ces variables, alors l'ensemble s_j correspondant couvre e_i. Donc si à l'ensemble des variables vidées on fait correspondre $S \subset \mathcal{S}$, S couvre bien E.

■

Ce cas (non réaliste au vu des architectures modernes) est donc NP-complet. Nous allons voir maintenant un cas plus réaliste où le nombre de trous est borné.

3.2.2 Diminuer le nombre de variables vivantes d'une constante

On se place dans le cas d'un graphe d'intervalles avec t trous dans un bloc de base, c'est-à-dire qu'il y a au maximum t utilisations de variables en un point. Le coût d'allouer en mémoire est quelconque et on cherche à passer de Ω à $\Omega - k$ avec k constante. En tout point du programme il faut donc qu'il reste au maximum $\Omega - k$ variables en vie.

Nous allons montrer que ce problème est polynomial car k est constante.

Dans la suite, on appelle *solution* un ensemble de variables tel que si on les vide toutes en mémoire, on est en tout point à au plus $\Omega - k$ variables en vie. On identifiera la variable avec son intervalle de vie dans les démonstrations.

Nous commençons par énoncer deux lemmes qui nous serviront pour prouver la polynomialité de notre algorithme.

Lemme 5. *En un point donné, $t + k$ variables quelconques forment une solution.*

Proof. Au point p, si l'on vide en mémoire $t + k$ variables quelconques, alors au plus t de ces variables sont utilisées donc au moins k variables seront effectivement vidées en ce point : le nombre de variables en vie en ce point diminuera d'au moins k. ■

Lemme 6. *Dans une solution optimale, il y a en tout point au plus $2(t + k)$ variables de la solution en vie en même temps.*

Proof. S'il existe un point p du programme tel que, en ce point, il y a strictement plus de $t + k$ variables en vie, on étend ce point en un intervalle d'instructions maximal tel que en tout point il y ait strictement plus de $t + k$ variables en vie. (Si p n'existe pas, le lemme est vrai).

Cet intervalle ne contient pas complétement l'intervalle de vie d'une variable de la solution sinon on pourrait l'enlever et d'après le lemme 5 le résultat serait toujours une solution or la solution était optimale. Donc touts les intervalles des variables débordent : ils dépassent soit à gauche soit à droite de l'intervalle. Or l'intervalle est maximal, donc à gauche et à droite il y a moins de $t + k$ variables en vie donc il y a en tout moins de $2(t + k)$ variables qui débordent. Il y a donc en tout point de l'intervalle au maximum $2(t + k)$ variables de la solution vivantes en même temps. ■

Définitions. Soit un intervalle d'instructions I (une séquence d'instructions consécutives). Soit s_i une solution, on note $I(s_i) = \{v \in s_i : v \subset I\}$ (variables qui ne dépassent pas de I). Deux solutions sont équivalentes sur I : $s_1 \sim s_2 \iff s_1 - I(s_1) = s_2 - I(s_2)$ (elles sont égales en dehors de I). On note $G(s) = \{v \in s : v \text{ dépasse à gauche } \}$ et $D(s) = \{v \in s : v \text{ dépasse à droite } \}$

Remarquons que $s = G(s) \cup D(s) \cup I(s)$, donc $s_1 \sim s_2 \iff G(s_1) = G(s_2)$ ET $D(s_1) = D(s_2)$. On note alors une classe d'équivalence de $\sim$: (G, D) puisque c'est l'intérieur, I, qui

différencie les solutions d'une classe d'équivalence. Si l'on note n le nombre de variables, le nombre de classes d'équivalence définies par $\sim$ sur I est inférieur à $(C_n^{2(t+k)})^2$ (choix des variables à gauche et à droite parmi $2(t+k)$ d'après 6) donc inférieur à $n^{4(t+k)}$.

Une solution s est minimale (pour I) si pour toute solution t, le coût de t est supérieur ou égal au coût de s (somme des poids des intervalles choisis).

On définit arbitrairement un représentant minimal pour chaque classe d'équivalence.

Lemme 7. *Pour chaque classe c de solutions pour l'intervalle $[p;r]$, on peut calculer en $O(n^{2(t+k)})$ un représentant minimal à partir des représentants minimaux pour $[p;q]$ et $[q+1;r]$ pour tout q point du programme de $[p;r[$.*

Proof. Soit une classe c de solutions de $[p;r]$. Pour tout $s \in c$, $D(s)$ et $G(s)$ sont fixés et notés D et G. Pour simplifier la lecture, posons $s_{|[p;q]} = [p;q](s)$.

Soit s minimale pour la classe (G,D). Alors soit $A = D(s_{|[p;q]}) = G(s_{|[q+1;r]})$ car un intervalle qui dépasse à droite en q dépasse à gauche en $q+1$. Alors (G,A) est une classe d'équivalence pour $[p;q]$ et (A,D) est une classe d'équivalence pour $[q+1;r]$. Si on modifie s en remplaçant $s_{|[p;q]}$ par un représentant minimal de (G,A) pour $[p;q]$, et de même sur $[q+1;r]$, on obtient une meilleure solution ce qui n'est pas possible de manière stricte. Donc une solution optimale s est constituée de représentants optimaux de (G,A) et de (A,D).

Mais pour construire s, on ne sait pas *a priori* quel A choisir. On les teste tous. Il y en a $C_n^{2(t+k)}$ et si on a stocké les calculs précédents, chaque test prend un temps constant. On peut donc bien le faire en $O(n^{2(t+k)})$.

■

Si l'on a n variables et m points de programme, alors nous avons donc un algorithme en complexité en temps $O(mn^{6(t+k)})$ qui nous trouve la solution optimale sur le programme.

Cependant l'algorithme trouvé n'est pas satisfaisant puisqu'on retrouve t et k en exposant ; il devient donc exponentiel si l'un ou l'autre de ces paramètres n'est pas une constante. Or c'est le problème $\Omega \searrow r$ qui nous intéresse le plus pour des applications pratiques.

3.2.3 Diminuer jusqu'à une variable

C'est le cas où l'on a un nombre fixé de registres. C'est le problème que nous devons résoudre si l'on veut implanter le vidage en mémoire dans un compilateur. Nous allons montrer que l'on ne peut *a priori* pas espérer trouver un algorithme où k n'est pas en exposant pour la partie précédente : si k n'est pas constant (ici $k = \Omega - r$ or Ω et r ne sont pas des constantes) alors le problème est NP-complet[4].

Nombre de trous $t \geq 2$. Soit le problème suivant :

Le nombre de trous t est fixé et supérieur ou égal à deux. Les poids sont tous égaux à un. Peut-on passer de Ω à la variable r pour un coût inférieur ou égal à R ?

[4]sauf peut-être si $t = 1$ et les poids sont unitaires, problème que nous n'avons pas résolu.

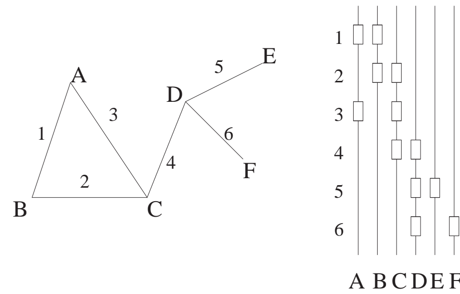


Figure 5: Réduction à *independent-set*.

Ce problème est NP-complet.

Rappelons le problème NP-complet du *independent-set* (problème [GT20] du Garey et Johnson [9]) : un graphe $G = (V, E)$ quelconque, avec n le nombre de sommets et m le nombre d'arêtes. Peut-on trouver un ensemble de sommets de taille R tel que deux sommets quelconques de cet ensemble ne soient jamais voisins dans G ?

Proof. Réduction de *independent-set* :

Soit un graphe G dont on désire trouver un ensemble indépendant de taille R. On crée une instance du problème de vidage en mémoire suivant : n variables toutes vivantes sur m points consécutifs.

Pour toute arête e du graphe, les deux variables correspondant aux sommets voisins sont utilisées au point e du programme. Elles et elles seules ont alors un trou à cet endroit (voir figure 5).

Alors il y a équivalence entre l'existence d'un ensemble indépendant de taille R et l'existence d'une solution de coût inférieur ou égal à R pour diminuer le nombre de variables vivantes à $n - R + 1$.

En effet, considèrons une solution de *independent-set* de taille R. Si l'on vide en mémoire les variables de l'instance correspondante, alors en chaque point du programme, le nombre de variables en vie diminue d'au moins $R - 1$ car en un point donné il y a au plus une des variables choisies qui a un trou. Donc on a bien réduit à $n - R + 1$ variables en vie en tout point pour un coût R, puisque chaque variable a un poids unitaire.

Réciproquement, soit une solution de vidage, alors son coût est supérieur ou égal à R : si on ne vide que $R - 1$ variables il existe un endroit où $n - (R - 1) + 1 = n - R + 2$ variables sont en vie à cause des trous. Soit une solution de coût inférieur ou égal à R, alors le coût est exactement R. Dans cette solution, on ne peut pas avoir choisi deux variables qui ont des trous au même endroit, car sinon en ce point il y aurait encore au moins $n - R + 2$ variables. Donc les sommets correspondants dans le graphe ne sont pas reliés par des arêtes, c'est donc un ensemble de sommets indépendants de taille R. ■

Nombre de trous $t = 1$ et poids non unitaires Pour traiter le dernier cas où le nombre de trous maximum par point du programme est 1, la réduction est toujours de *independent-set* mais pour chaque arête, on met deux trous successifs sur les variables correspondantes et on

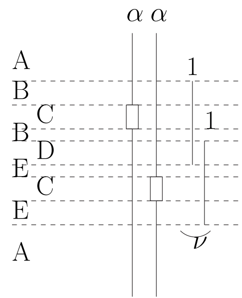


Figure 6: Réduction avec un seul trou par point.

ajoute deux petites variables ν qui recouvrent les trous, de coût faible (égal à 1) par rapport au coût des variables (α bien choisi) (voir figure 6 : les lettres correspondent à des zones et sont utilisées plus tard dans la démonstration).

a) les intervalles ν sont non troués. Nous le supposons pour simplifier en premier lieu la démonstration. Nous expliquerons en b) comment modifier la preuve pour qu'elle reste valide si les intervalles ν sont troués.

Les instances sont les même que pour la réduction précédente : aux n sommets correspondent n variables et à un ensemble indépendant de taille R correspond un choix des variables pour diminuer Ω à $n - R + 1$.

Proof. Preuve de la réduction : montrons qu'il existe un ensemble indépendant de taille R si et seulement si il existe une solution au problème de vidage de coût inférieur ou égal à $\alpha R + m$.

Soit V_R un ensemble indépendant de taille R, alors on obtient une solution à notre problème en choisissant les R variables correspondantes ainsi que pour chaque arête du graphe un des deux ν : si l'un des deux sommets est dans V_R, le ν correspondant au trou de celui-ci, sinon, l'un des deux ν au hasard.

Montrons que le nombre de variables en vie sur chaque type de zone est bien plus petit que $n - R + 1$:

- Si aucune des deux variables n'est dans V_R, supposons par symétrie que c'est le ν le plus haut qui est choisi dans la solution. Comptons le nombre de variables en vie dans chaque zone :

  A. $n - R$, ok ;
  B. $n + 1 - R - 1$, ok ;
  C. $n + 1 - R - 1$, ok ;
  D. $n + 2 - R - 1$, ok ;
  E. $n + 1 - R$, ok ;

- Si une des deux variables est dans V_R, supposons par symétrie que ce soit celle qui a le trou le plus haut :

  A. $n - R$, ok ;
  B. $n + 1 - R - 1$, ok ;
  C. $n + 1 - (R - 1) - 1$, ok ;
  D. $n + 2 - R - 1$, ok ;
  E. $n + 1 - R$, ok ;

Un ensemble indépendant fournit donc bien une solution à notre problème de vidage et le coût de cette solution est $\alpha R + m$ avec m le nombre d'arêtes du graphe.

Réciproquement, soit une solution au problème de vidage de coût inférieur ou égal à $\alpha R+m$. Voyons le nombre de variables (pas les ν) qu'elle a vidées :

- une solution avec $k < R$ variables n'est pas possible à cause de la zone C : soit une zone C où l'on a vidé l'intervalle qui possède le trou, alors le nombre de variables en vie est de $n - (k - 1) \geq n - R + 2 > n - R + 1$;
- si notre solution comporte $k \geq R + 1$ variables, alors son coût est supérieur ou égal à $\alpha k = \alpha R + \alpha$. Nous n'avons pas encore défini α, nous choisissons donc $\alpha = m + 1$ et ainsi, la solution a un coût strictement supérieur à $aR + m$. Contradiction

Donc notre solution a vidé exactement R variables.

Reste à montrer que ces variables ne sont pas associées à des sommets voisins. Supposons par l'absurde que ce soit le cas, et observons l'état des zones de la figure 6 pour l'arête concernée (la soustraction $-\nu$ signifie -1 si ν est vidé, -0 sinon) :

A. $n - R$, ok ;

B. $n + 1 - R - \nu_1$, ok ;

C. $n + 1 - (R - 1) - \nu_1$, donc ν_1 doit être vidé en mémoire ;

D. $n + 2 - R - \nu_1 - \nu_2$, ok ;

E. $n + 1 - (R - 1) - \nu_2$, donc ν_2 doit aussi être vidé en mémoire ;

Donc il ne reste plus que $m - 2$ intervalles ν pouvant être choisis sur les $m - 1$ arêtes restantes. Or la zone D implique que pour n'importe quelle arête, au moins un des deux ν doit être vidé en mémoire car sinon elle comporte $n + 2 - R$ variables en vie. Donc il y a au moins $m - 1$ intervalles ν de plus choisis, un par arête existante, ce qui nous fait au total un coût supérieur à $\alpha R + m + 1$ ce qui est contradictoire.

Ainsi, si notre solution a un coût égal à $\alpha R + m$, elle ne contient jamais deux sommets voisins et donc à l'ensemble des variables choisies correspond un ensemble indépendant du graphe initial. ■

b) les intervalles ν sont troués : en réalité, les intervales ν correspondent à des variables et donc ne deviennent vivantes qu'après leur définition, et meurent après leur dernière utilisation. Cela implique que le domaine de vie d'un ν commence et se termine par un trou. La preuve doit donc être modifiée pour être correcte. Nous donnerons seulement l'idée pour la création de l'instance : il faut rajouter deux variables par ν dont les domaines de vie forment le complémentaire du ν sur le bloc de base. Leur poids est très gros si bien qu'ils ne sont jamais choisis pour être vidés en mémoire. Ainsi, quand la première des deux variables meurt, elle libère un registre qui peut être utilisé lors de la définition du ν pour le stocker.

3.3 Conclusion

Le problème du vidage sous SSA est donc plus compliqué que l'on pourrait penser. Nous avons montré que pour avoir une modélisation exacte du problème il nous fallait introduire la notion de trou ; le problème est alors NP-complet dans la plupart des cas. Nous avons trouvé un cas de polynomialité mais qui se révèle inutilisable en pratique car exponentiel pour les cas qui nous intéressent en allocation de registres. Le point positif de cette étude est que c'est la première fois qu'une étude de complexité est faite sur le vidage en mémoire seul (séparé de l'allocation de registres). Cette étude nous a permis de mieux comprendre ce problème notamment au niveau des trous. En particulier les résultats obtenus montrent que la complexité se trouve soit dans la différence entre le nombre maximum de variables en vie Ω et le nombre de registres disponibles r, soit dans le nombre de registres disponibles. Ainsi, dans des architectures comme le pentium (8 registres entiers) ou à l'inverse dans le st200 (64 registres), une approche combinatoire est envisageable. De même à propos des trous, t est en pratique petit ($t \leq 3$) ce qui rend le problème de vidage très probablement approximable. Notons que ces travaux n'annulent pas les intérêts de la forme SSA et nous espérons pouvoir tirer tout de même parti de ses particularités ; par exemple la structure d'arbre du graphe de dominance est très importante et laisse entrevoir des possibilités d'algorithmes de type programmation dynamique qui travailleraient des feuilles vers la racine.

Il reste toujours le problème du retour de la forme SSA : cette forme n'est qu'une représentation intermédiaire de travail et il faut revenir à une forme standard après son utilisation. Nous verrons dans la partie suivante que ce retour pose problème : de nombreuses instructions de copie (*move*) doivent être ajoutées au code ce qui réduit son efficacité. Aussi-avons nous cherché, dans le but de l'utiliser conjointement avec une implantation du vidage en mémoire sous SSA, un moyen de limiter l'ajout de ces fonctions de copie.

4 Implantation

Nous présenterons dans cette partie une solution pratique au problème de la forme SSA annoncé dans la conclusion précédente (3.3).

Dans le cadre de la collaboration de l'équipe Compsys avec STMicroelectronics, la partie implantation a été réalisée dans le LAO2 de l'équipe MCDT (*Micro Core Development Tools*). Ce compilateur a pour nom *Linear Assembly Optimizer* et est en réalité une bibliothèque de compilation. Cette bibliothèque est alors utilisée par le compilateur *open-source* Open64 ; le but est de considérer dans la bibliothèque des optimisations spécifiques à l'architecture du

processeur st200 de chez STMicroelectronics.

4.1 Motivations

Un des buts de l'implantation était de se familiariser avec ce compilateur, écrit en *xcc*, un sur-langage de C développé en interne par l'équipe MCDT et utilisé exclusivement pour le LAO2.

Nous avons de modifié l'allocateur de registre qu'avait implanté Cédric Vincent l'année précédente. Le but était de raffiner le graphe d'interférence en supprimant certaines arêtes inutiles. L'idée part d'une définition plus exacte de la notion d'interférence que nous avons remarqué : deux variables interfèrent si et seulement si leurs domaines de vie sont d'intersection non nulle *et* leurs valeurs diffèrent aux points d'intersection . En effet, si deux variables interfèrent mais ont la même valeur quand elles coexistent, il est possible de les assigner sans danger au même registre.

La motivation pour utiliser cette définition est le passage par la forme SSA qui crée beaucoup d'instructions *move* et génère donc des interférences artificielles entre les variables. Étudions l'exemple de la figure 7 : incrémenter un compteur dans une boucle. Le passage par la forme SSA interdit la redéfinition de la variable a dans la boucle, on procède donc à un renommage des variables : la première sera a_1. En entrée de la boucle, la valeur de a_2 peut provenir soit de l'initialisation, soit d'une itération précédente de la boucle ce qui est représenté à l'aide de la fonction ϕ. Les fonctions ϕ ne correspondent à aucune instruction réelle : c'est juste une notation qui indique qu'en fonction de la provenance du flot d'instructions, on sélectionne telle ou telle valeur. Dans notre cas, $\phi(a_1, a_2)$ signifie a_1 si on vient de l'initialisation de la boucle, a_2 si l'on vient d'une itération précédente de la boucle . L'incrément lui-même est alors fait sur un troisième a_3. Enfin, La valeur en sortie provient soit de l'initialisation, soit de la boucle d'où un a_4 unificateur défini par une fonction ϕ.

Après avoir étudié la forme SSA, on repasse sous forme standard, les fonctions ϕ sont remplacées par des copies (*move*) à la fin des blocs prédécesseurs pour donner à a_2 et a_4 leurs valeurs réelles. Le compilateur s'aperçoit que a_1 et a_4 n'interfèrent pas et donc renomme a_4 en a_1. Par contre il n'arrive pas à simplifier plus car a_1 interfère avec a_2 et a_3. Or il est évident que toutes ces variables contiennent la même valeur dès lors qu'elles sont vivantes en même temps. On pourra noter qu'en l'absence de transformations faites sous SSA il est évident que les a_i peuvent être assignés au même registre ; en pratique, les optimisations faites sous SSA cassent cette propriété.

Le but est donc de calculer statiquement[5] les valeurs des variables en tout point du programme. Lors de la création du graphe d'interférences, on vérifie avant d'ajouter une arête que les deux variables correspondantes ne possèdent pas la même valeur.

4.2 Analyse des valeurs

Pour connaître la valeur d'une variable en un point du programme, il nous faut connaître toutes les valeurs possibles de cette variable aux points prédécesseurs. Le graphe de contrôle

[5] à la compilation et non à l'exécution

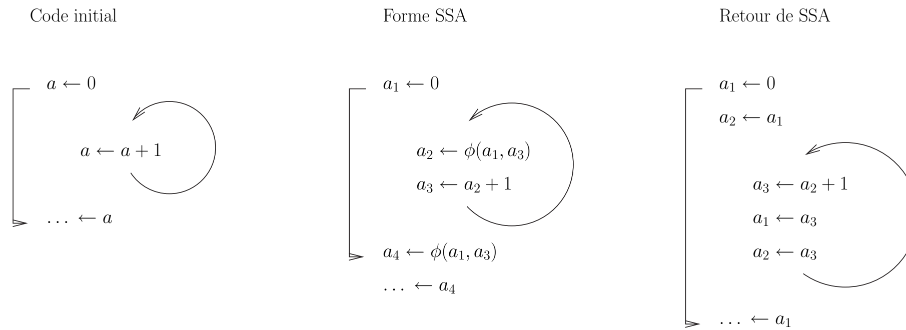


Figure 7: Exemple de passage en SSA

de flot est orienté mais quelconque, il est donc possible qu'il y ait des cycles. Une manière simple de traiter ce problème est d'avoir recours à un algorithme de point fixe. Comme la valeur en un point dépend des points précédents, il nous faut parcourir les blocs de base de la première instruction à la dernière (parcours *top-down*, du haut vers le bas). Par contre, la création du graphe d'interférences, qui utilise aussi un algorithme de point fixe, s'effectue par un parcours de bas en haut (*bottom-up*). On utilise alors trois passes au lieu d'une seule :

1. parcours de haut en bas : calcul des valeurs ;
2. parcours de bas en haut : calcul des interférences ;
3. parcours de haut en bas : ajustement des interférences en fonction des valeurs.

Nous allons maintenant commencer par définir l'analyse de flot de données (*data flow analysis*) qui nous donnera les informations souhaitées sur les valeurs des variables.

4.2.1 Aspect théorique

Soit V l'ensemble des variables du programme. Soit f la fonction d'état définie en un point du programme par le couple (d, e) avec :

- d la fonction de définition : $V \rightarrow \{\top, \nu, ?\}$; $\top$ signifie non définie , ν est la valeur de la variable si elle est connue et ? signifie que la variable est définie mais de valeur inconnue ;
- e la fonction d'égalité : $V \times V \rightarrow \{0, 1\}$ où 1 signifie que les variables sont égales si elles sont toutes deux définies.

Remarque : d'après la définition de e, si $d(a) = \top$ alors pour toute variable b, $e(a, b) = 1$. Cette notation est justifiée par le fait qu'une variable non définie peut être assignée au même registre que toute autre variable : soit on n'a pas besoin de cette variable, soit le programme initial est faux.

Il nous faut maintenant définir deux modificateurs de fonction d'état :

- la loi de jonction (*join*) $\wedge$ sert à définir la fonction d'état à la jonction de chemins du graphe de flot de contrôle : si un bloc de base B a pour prédécesseurs B_1 et B_2, dont les fonctions d'états en fin de bloc sont f_1 et f_2, alors la fonction d'état en début de B est $f = f_1 \wedge f_2$;

- la fonction de transfert F : dans un bloc de base, calcule la fonction d'état après une instruction en fonction de f avant l'instruction.

Propriétés des modificateurs : les modificateurs doivent posséder certaines propriétés pour garantir l'existence d'une solution et la convergence de l'algorithme de point fixe. Kildall les présente dans son article de 1973 sur les optimisations de programme [12] :

- Définissons une relation d'ordre sur les fonctions d'état : $f \leq g \Leftrightarrow f \wedge g = f$; pour que $\leq$ définisse un treillis, il faut que $\wedge$ soit associative, commutative et idempotente ;

- si F est monotone ($f \leq g \Longrightarrow F(f) \leq F(g)$) alors il existe un point fixe ; de plus, le point fixe maximal est atteint quel que soit l'ordre de parcours des blocs de base ;

- si F est distributive ($F(f \wedge g) = F(f) \wedge F(g)$), alors l'intersection de toutes les fonctions d'état obtenues par tous les chemins possibles est un point fixe ; c'est celui-là qui est atteint.

Choix des modificateurs : nous avons défini la fonction $\wedge$ par $(d, e) \wedge (d', e') = (d \wedge d', e \wedge e')$ avec :

- $(e \wedge e') = e$ ET e' où ET est la conjonction binaire ;

- $(d \wedge d')$ est représenté pour une variable a dans le tableau suivant :

$d(a) \backslash d'(a)$	$\top$	ν	$\mu \neq \nu$	?
$\top$	$\top$	ν	μ	?
ν	ν	ν	?	?
?	?	?	?	?

Pour la fonction de transfert F, $F(d, e) = (F(d)(e), F(e)(d))$ (la modification de d dépend de e et réciproquement) et elle ne modifie son argument que dans les cas des trois instructions suivantes :

- affectation : `a` $\leftarrow$ ν . La variable devient définie avec la valeur ν et est égale aux variables de même valeur.

$$
\begin{aligned}
F(d)(a) &= v \\
F(e)(a, b) &= 1 \text{ si } d(b) = \nu \text{ ou } \top \\
&= 0 \text{ sinon}
\end{aligned}
$$

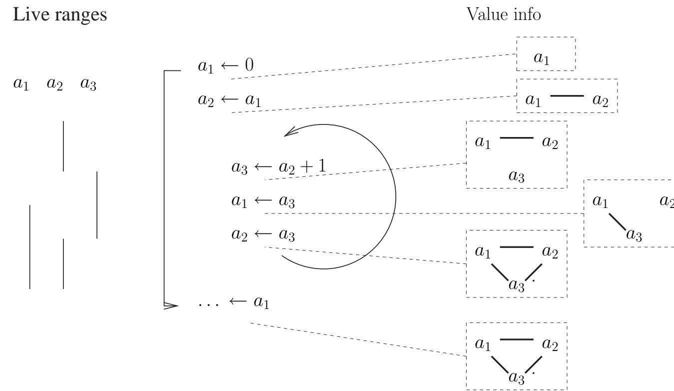


Figure 8: Motivation pour l'étude de valeurs

- copie : `a ← b` . a prend toutes les propriétés de b : valeur donc égalités.

$$\begin{aligned} F(d)(a) &= d(b) \\ F(e)(a,c) &= e(b,c)\ \forall c \end{aligned}$$

- définition : `a ← ...` a est définie mais avec une valeur inconnue

$$\begin{aligned} F(d)(a) &= ? \\ F(e)(a,b) &= 1 \text{ si } d(b) = \top \\ &= 0 \text{ sinon} \end{aligned}$$

Nous avons montré que toutes les propriétés sont bien vérifiées par les fonctions choisies. Les démonstrations étant fastidieuses et inintéressantes, nous ne les avons pas explicitées ici.

La fonction d'égalité e calculée est montrée en tout point de l'exemple sur la figure 8 sous la forme d'un graphe dont les sommets sont les variables et les arêtes les couples dont l'image par e est 1.

4.2.2 Aspect pratique

Dans cette partie, nous discutons des choix effectués en pratique pour l'implantation, on y trouve la structure de donnée utilisée, les optimisations pour une bonne utilisation de la mémoire et la méthode de travail qui nous garantit une complexité linéaire en temps.

Structures de données : une fonction d'état sera codée naturellement par un objet à deux champs :

- pour la fonction d, il est inutile de stocker les variables qui ne sont pas définie à un point donné. On utilise donc une table de hachage dans laquelle, en fonction de $d(a)$:

$\top$: la clé a n'existe pas ;

? : la clé a est associée à un couple `{ Unknown, 0 }`

ν : la clé a est associée à un couple `{ Value,` ν `}`

- pour la fonction e : celle-ci définit un graphe mais contenant très peu d'arêtes. Une table de hachage est également utilisée : les clés sont des couples de variables et seules les égalités sont présentes dans la table. Il n'y a pas de valeur associée aux clés puisque la simple présence dans la table contient suffisamment d'informations.

Gestion de la mémoire : le principe de l'analyse de flot de donnée est d'itérer sur tous les blocs de base jusqu'à atteindre un point fixe. Or nous avons besoin de connaître les fonctions d'états entre toutes les instructions du programme ce qui pose un sérieux problème de mémoire. Aussi nous avons décidé de ne stocker qu'une seule fonction d'état par bloc de base : celle après la dernière instruction du bloc. Nous avons choisi la dernière du bloc car pour calculer la fonction d'état du début d'un bloc, il est nécessaire de connaître les fonctions d'état des fins des blocs prédécesseurs afin de leur appliquer $\wedge$. Ces calculs sont indispensables pendant la phase de stabilisation. Il est ensuite aisé (linéaire) de connaître la fonction d'état f en tout point du bloc en appliquant successivement F à partir du début.

Utilisation : supposons que nous ayons effectué la première passe et donc nous connaissons pour chaque bloc de base la fonction d'état de la fin du bloc. Il nous faut maintenant construire le graphe d'interférence, en prenant soin d'éviter d'ajouter des arêtes inutiles.

Le problème est le suivant : la méthode de construction du graphe *doit* parcourir chaque bloc de base de la dernière instruction à la première car ce sont les utilisations qui permettent de savoir que les variables sont vivantes *avant* ; par contre pour connaître la fonction d'état en un point on *doit* parcourir le bloc de base de la première instruction à la dernière car ce sont les définitions qui permettent de connaitre les valeurs des variables *après*.

Pour éviter de tomber dans une complexité quadratique en temps (en recalculant pour chaque arête à ajouter dans le graphe la fonction d'état à partir du début du bloc) en temps, ou occuper potentiellement énormément de mémoire (en mémorisant la fonction d'état de chaque point du bloc de base), l'ajout des arêtes se fait en deux phases :

1. parcourir le bloc de base du bas vers le haut et *empiler* les arêtes d'interférences dans une pile au lieu de les ajouter au graphe ;

2. parcourir le bloc de base du haut vers le bas en calculant la fonction d'état, et à chaque instruction où des arêtes ont été empilées, les dépiler et les ajouter au graphe uniquement si les variables concernées ont des valeurs différentes en ce point.

Ceci garantit que l'espace mémoire occupé est linéaire en le nombre d'arêtes à ajouter dans un bloc de base (une arête occupe beaucoup moins de place d'une fonction d'état) et le temps de parcours est linéaire en la taille du bloc de base.

4.2.3 Résultats

Nous avons testé l'algorithme implanté sur la suite de programmes de tests utilisée par l'équipe MCDT de STMicroelectronics pour valider les optimisations sur leur compilateur de production. Ces programmes sont donc choisis pour être représentatifs des applications ciblées par le processeur st200. Nous avons dans un premier temps testé l'algorithme sur des codes qui ne sont pas passés par la forme SSA : ces codes sont très fortement optimisés et l'algorithme ne peut y gagner que très peu, ce qui a facilité le déboguage. Nous avons alors activé le passage par la forme SSA et obtenus les résultats du tableau suivant :

	Sans SSA	SSA	SSA_{opt}	gain
nombre de copies	203	276	212	**87 %**
coût des copies	11129	18884	12014	**88 %**
coût du vidage	332676	291354	291354	**12 %**
coût total	343805	310238	303368	**11 %**

La première colonne donne les valeurs pour des programmes dans lesquels la forme SSA n'a pas été activée. Dans la deuxième colonne, les programmes ont été mis sous forme SSA puis ont été sortis de SSA sans optimisation particulière. Dans la dernière colonne, nous avons activé notre algorithme de réduction des copies après le retour de la forme SSA. Les lignes de coût représentent le nombre de cycles processeur utilisés pour les copies et le vidage en mémoire lors de l'exécution des programmes tests. L'expérience montre l'utilité de l'algorithme implanté et il reste à étudier au cas par cas les copies supplémentaires qui n'ont pu être éliminées pour en déterminer la cause.

4.3 Améliorations

Nous ne parlerons pas ici des amélioration évidentes qui peuvent être faites pour tenir compte de plus d'instructions (par exemple l'addition) lors de l'analyse de valeurs : il existe énormément d'articles sur ce sujet et nous avons justement choisi ici la simplicité d'un algorithme approprié à nos besoins.

Par contre une amélioration intéressante serait d'utiliser la certitude d'existence de variables pour déterminer la valeur d'autres variables. Un exemple s'impose. Considérons le graphe de contrôle de flot de la figure 9.

La question est de savoir ce que vaut la variable b en dans le bloc F. Elle est définie dans le bloc C avec la valeur 4 ou 5 selon que l'on vient du bloc A ou du B, et dans le bloc E avec la valeur 5. Avec l'analyse de valeurs présentée ci-avant, $d(b) =?$ dans le bloc F.

Cependant, la valeur de b n'est pas quelconque dans le bloc D...

- si l'on n'est pas passé par le bloc C, b est non défini : alors en F la fonction $\wedge$ fusionne le non défini avec la valeur 5 ; $d(b) = 5$;
- si l'on est passé par le bloc C, b a la valeur de a. Or dans le bloc D, la variable c est utilisée : ce n'est possible que si l'on est passé par le bloc A puisque le chemin venant de B ne définit pas c. Donc $d(a) = 5$ or b a été initialisé avec a donc $d(b) = 5$.

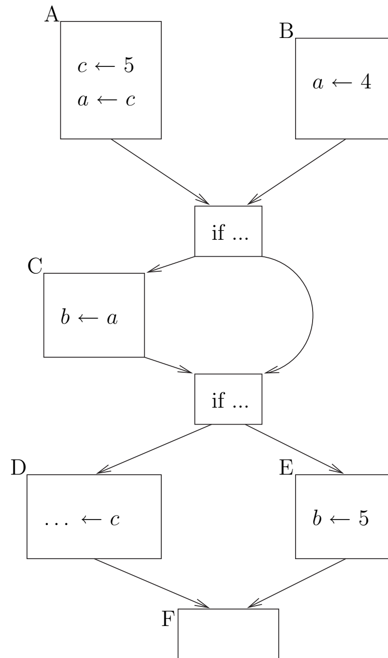
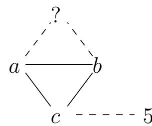


Figure 9: L'existence de c nous informe de la valeur de b.

Ainsi, b ne peut valoir que 5 dans le bloc F si elle est définie.

Application à l'analyse de valeur : ce raisonnement à l'aspect compliqué est en réalité extrêmement logique dans le cadre de l'algorithme d'analyse présenté précédemment : en entrée du bloc D la fonction d'état est la suivante :



Rappelons la règle de la fonction e qui correspond à l'existence d'arêtes : s'il y a une arête entre v et w c'est que les variables sont égales si elles sont toutes deux définies . Il est facile de voir qu'en D a et c sont définies (D est dominé par A et B qui définissent a, et c est utilisé dans D). Donc a a la valeur 5 et dans le graphe, le '?' est remplacé par 5. Comme il était commun à a et b, cette dernière a donc aussi la valeur 5.

Il faudrait donc modifier l'algorithme pour que les variables puissent pointer vers le même '?' ; et ajouter un calcul d'existence (et non-existence) des variables dans les blocs de base. Ce dernier point n'est pas possible pour toutes les variables en tout point du programme mais il existe des cas où l'on peut affirmer qu'une variable est ou n'est pas en vie en un point précis du programme (par exemple dans un bloc de base entre la dernière utilisation avant la prochaine définition).

Pour cela remarquons par exemple qu'une variable v est nécessairement en vie en un point P si :

- tout chemin de la racine à P contient une définition de v ;
- tout chemin de P à la sortie passe par une utilisation de v sans redéfinition intermédiaire.

Il suffit alors de vérifier ces conditions par des algorithmes de parcours de graphe *pre*-fixes ou *post*-fixes de type *reaching-definitions* (chapitre 17 du *Tiger book* [1]).

5 Conclusion

Dans ce rapport nous nous sommes intéressés au problème du vidage en mémoire lors de la compilation de code. Le cas général du choix des variables à vider est NP-complet et connu depuis longtemps. Nous nous sommes alors intéressés à la représentation de code particulière : la forme SSA. Nous avons montré que les propriétés de cette représentation rendaient le coloriage du graphe d'interférence facile car celui-ci devient triangulé. Notre étude de complexité du problème de vidage sous SSA nous a mené à la définition des trous ce qui permet de modéliser exactement le problème du vidage. Nous avons alors prouvé que, même dans des cas simples comme le bloc de base, le problème est souvent NP-complet. C'est donc une étude théorique de problèmes que l'on croit connus depuis longtemps car les modèles sont un peu flous, et il s'est révélé qu'en fait il n'existe pas d'algorithme optimal utilisable en pratique.

L'inconvénient du passage par la forme SSA est la création de nouvelles variables et de copies (instructions *move*) qui augmentent la taille du code et réduisent son efficacité. Aussi nous avons décidé d'affiner le graphe d'interférence en détectant des arêtes inutiles ce qui permet ensuite de supprimer des copies. La méthode se base sur une analyse de valeurs et d'égalités de variables. L'implantation a été réalisée sur le compilateur de production de l'équipe MCDT de STMicroelectronics utilisé pour leur processeur st200.

L'étude de complexité que nous avons menée nous a permis de bien comprendre le problème du vidage en mémoire. Nous pensons que les propriétés de la forme SSA vont nous permettre de trouver de bonnes heuristiques pour le vidage. Ceci nous permettrait alors d'avoir une nouvelle méthode d'allocation de registres : vidage en mémoire d'abord, coloriage facile sous la forme SSA puis réduction du nombre de copies par des techniques classiques (*coalescing*) améliorées par notre définition plus fine d'interférence.

References

[1] Andrew w. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[2] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[5] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[7] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27. IEEE Computer Society Press, August 1987.

[8] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[10] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57).* North-Holland, 2004.

[11] Kenneth Zadeck Kathleen Knobe. Register allocation using control trees. 1992.

[12] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM Press.

[13] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *Proceedings of the conference on Programming language design and implementation*, pages 268–277. ACM Press, 1993.

[14] Mihalis Yannakakis. Node-and edge-deletion np-complete problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing (STOC)*, pages 253–264, 1978.