Laboratoire de l'Informatique du Parallélisme

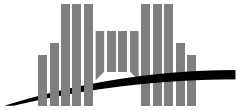
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Stage en laboratoire, allocateur de registres

Cédric Vincent 5 Avril 2004

Rapport de Maîtrise Informatique

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



Stage en laboratoire, allocateur de registres

Cédric Vincent

5 Avril 2004

Laboratoire et entreprise:
- Laboratoire de l'Informatique du Parallélisme à l'École Normale Supérieure de Lyon ;
- STMicroElectronics.

Responsable:
- Alain Darte, CR CNRS

Tuteurs:
- Benoît Dupont De Dinechin, ST ;
- Christophe Guillon, ST ;
- Fabrice Rastello, CR INRIA ;

Résumé: Ce rapport se divise en deux parties :
- la première expose les caractéristiques de l'environnement du stage, ainsi que mon expérience humaine ;
- la seconde présente le travail effectué.

Ce rapport se veut le plus personnel possible.

Mots clefs: stage, maîtrise, laboratoire, compilation, allocation de registres, coloration de graphes, fusion de variables.

Table des matières

Chapitre 1

Expériences personnelles

1.1 Le laboratoire

1.1.1 Présentation

Ce stage se déroula à l'École Normale Supérieure de Lyon, dans le Laboratoire de l'Informatique du Parallélisme. Celui-ci est, en plus de l'ENS-Lyon, associé au CNRS, à l'INRIA et à l'UCB Lyon1, et est composée de sept projets dont CompSys (Compilation et systèmes enfouis) où je fis mon stage. Le groupe CompSys est de plus un projet de l'INRIA dont l'objectif est le développement de techniques d'optimisations spécifiques au processus de compilation pour les systèmes embarqués.

1.1.2 Membres de CompSys

L'équipe CompSys est composé actuellement de

quatre chercheurs :
- Tanguy Risset, CR1 INRIA, responsable scientifique ;
- Alain Darte, CR1 CNRS, mon maître de stage ;
- Fabrice Rastello, CR2 INRIA, ancien ingénieur de STMicroElectronics ;
- Paul Feautrier, Professeur ENS.

deux collaborateurs extérieurs :
- Anne Mignotte, Professeur Insa-Lyon ;
- Antoine Fraboulet, Maître de conférence, Insa-Lyon.

un doctorant :
- Antoine Scherrer.

un stagiaire :
- moi-même :)

1.2 L'entreprise STMicroElectronics

1.2.1 Présentation

Durant mon stage, j'ai développé avec, et pour les outils de l'équipe *Micro Core Development Tools* de chez STMicroElectronics. Cette équipe, dirigée par Christian Bertin, développe le compilateur Linear Assembly Optimizer 2 (LAO 2) pour le processeur ST 200 [1], dans lequel mon maître de stage et moi avons intégré un allocateur de registres.

[1] Pour plus de précisions sur ces termes techniques, se référer aux sections 2.1.1 et 2.1.2.

1.2.2 Visites

J'ai eu le plaisir d'être accueilli deux journées complètes chez STMicroElectronics, à Grenoble. La première fois, durant la première semaine de stage, les membres de l'équipe MCDT m'ont présenté leur entreprise ainsi que leur technologie LAO 2. La seconde fois, lors du dernier jour de stage, nous avons fusionné notre travail et le leur.

J'ai donc observé comment travaille une équipe de recherche et développement dans le domaine privé, et j'ai remarqué qu'il y avait beaucoup de similitudes avec le secteur public [2].

1.2.3 La philosophie du libre

L'une des premières choses que j'ai remarquée en arrivant sur le lieu de travail de l'équipe MCDT fut la présence de signes relatifs au mouvement du Logiciel Libre [3], notamment une fameuse affiche « Beware of the GNU ». Cela fut confirmé lorsque j'ai commencé à développer, puisque la technologie LAO 2 est sous licence GNU GPL [4] et GNU LGPL [5].

Il semblerait que cette philosophie soit partagée par l'entreprise entière puisque, lors de ma dernière visite, j'ai appris que des développeurs Anglais de chez ST avaient *porté* le noyau Linux pour leur processeur ST230.

Je fus donc doublement motivé pour travailler sur leurs technologies !

1.3 Motivations

1.3.1 Laboratoire

J'ai l'ambition de devenir enseignant-chercheur, c'est donc tout naturellement que mon choix s'est porté sur un stage en laboratoire. Je connaissais déjà deux chercheurs à l'ENS, Loïc Prylli au LIP [6] et Phillipe Saadé [7] de l'équipe PR@TIC. C'est pour cela que je fis une demande de stage auprès du LIP.

J'ai pu participer de manière active à la vie en laboratoire puisque j'ai assisté à des soutenances de thèses, des conférences, aux réunions périodiques du groupe CompSys, et même au conseil de laboratoire du LIP. J'ai pu assister à un cours de Fabrice Rastello dans le cadre du cours [12] de Tanguy Risset et à un cours d'Alain Darte dans lequel ses élèves effectuaient une présentation de leurs travaux de recherche.

1.3.2 Domaine de recherche

Cela fait déjà quelques temps que je m'intéresse aux systèmes embarqués, c'est-à-dire aux systèmes informatiques destinés à fonctionner dans un véhicule ou un appareil portable, comme une voiture, un PDA, ... J'apprécie le fait que les ressources soient limitées et qu'il soit nécessaire d'avoir une certaine rigueur. En effet, rien ne me déroute plus que les programmes gaspillant des méga-octets et des méga-hertz au nom de l'inutilité !

J'avais déjà une certaine expérience dans ce domaine puisque la plupart de mes Travaux Pratiques de Licence et Maîtrise sont effectués sur un Compatible IBM-PC avec un processeur Intel 486 DX2 à 60MHz, équipé de 7Mo de mémoire vive. Aujourd'hui une montre peut

[2] se référer à tout ce que j'ai observé au sein du LIP.
[3] http ://www.gnu.org/philosophy/philosophy.fr.html
[4] http ://www.gnu.org/copyleft/gpl.html
[5] http ://www.gnu.org/copyleft/lgpl.html
[6] mon encadrant de projet TER pendant l'année de Licence Informatique.
[7] mon encadrant de stage pendant la première année de DEUG MIAS.

aisément contenir cet équipement ! De même, je développe pendant mon temps libre pour Linux sur DreamCast [8] et sur GameCube [9].

1.4 Déroulement du stage

1.4.1 Calendrier

La figure 1.1 représente le diagramme du temps que j'ai passé sur différentes parties du stage. Il respecte plus ou moins ce que je m'étais fixé.

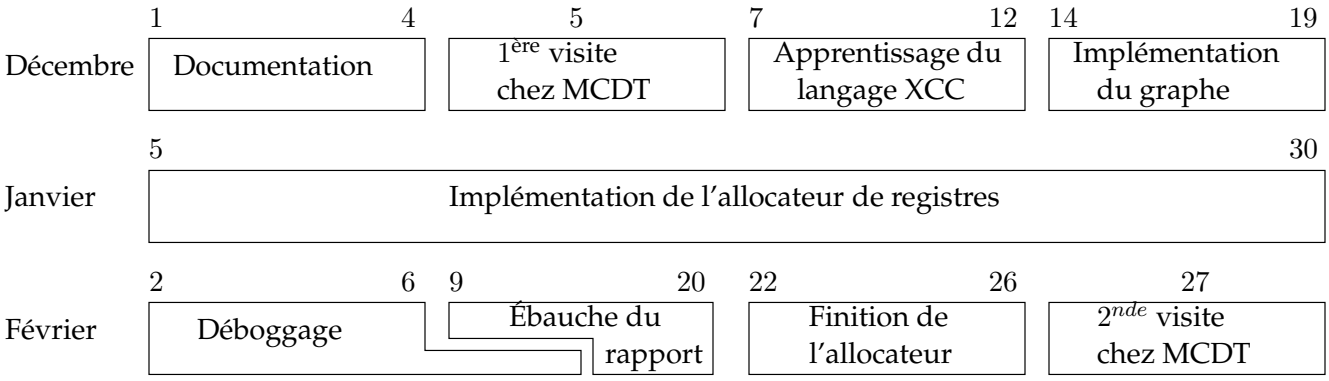


FIG. 1.1 – Diagramme de l'organisation du stage

Si je fais mon autocritique, je m'aperçois que j'ai essayé d'être trop productif le premier mois et qu'une semaine de plus à étudier mon sujet aurait été plus judicieux. En conséquence, le dernier mois fut un peu moins productif… J'ai donc atteint mon rythme de croisière au milieu du stage, durant Janvier.

1.4.2 Projet de recherche

C'est la première fois que je travaille sur un projet aussi important, et qui plus est en collaboration avec une entreprise. J'ai pu observé différentes techniques de conduite de projet et certaines méthodes de production.

De même, je me suis rendu compte qu'un tel travail nécessite énormément de recherche de documentation, non seulement pour savoir ce qui a été fait et ainsi servir de base à son travail, mais aussi pour trouver ce qui n'a pas été observé et donc apporter sa contribution.

1.4.3 Problèmes rencontrés et solutions

Un langage unique

Comme on le verra à la section 2.1.3, j'ai dû développer dans un nouveau langage, le XCC. Même si celui-ci m'a *énormément* facilité le développement grâce à l'automatisation de certaines tâches, je regrette tout de même une chose : c'est un langage utilisé uniquement dans la programmation du LAO 2 ! Cela est dû au fait qu'il soit développé en interne par l'équipe MCDT de ST et qu'aucune documentation n'ait été écrite à son sujet, ce qui fut un grand handicap… J'espère vraiment qu'il y aura un effort pour populariser ce langage !

[8] processeur Hitachi SH-4 128bits à 200Mhz, 16Mo de mémoire vive, sans disque dur.
[9] processeur IBM PowerPC Gekko 128bits à 405MHz, 40Mo de mémoire vive, sans disque dur.

Le rapport

La rédaction de ce rapport fut aussi un problème, puisque je traite d'un problème complexe. De plus, il est très difficile de ne pas rédiger en *Franglais* car certains termes techniques Anglais perdent leur pertinence en Français.

La structure de graphe

Le programme que nous avons écrit se base sur la notion formelle de *graphe*, il fallait donc que sa représentation informatique soit judicieusement choisie pour coller parfaitement aux algorithmes utilisés. J'ai dû écrire pas moins de quatre ébauches de structures de graphe, pour finalement aboutir à celle utilisée dans la documentation qui me servit de référence [1]. J'ai donc compris qu'il faut toujours comprendre le choix d'une implémentation en l'analysant soit pour la remettre en cause s'il y a un problème, soit pour l'utiliser et donc l'approuver.

1.4.4 Acquis

Outils libres

Cette année, durant les cours de Génie Logiciel, nous avons vu qu'il existe de nombreux outils libres pour faciliter et accélérer le développement, j'ai pu me perfectionner dans l'utilisation de quelques uns d'entre eux :

Doxygen : Ce logiciel [10] permet de générer de la documentation au format HTML, LaTeX,...

CVS : C'est un système [11] de contrôle de version pour les projets de développement.

Valgrind : Ce programme [12] permet de déboguer des programmes, il peut automatiquement détecter beaucoup de bogues de mémoire. Comme on peut le voir sur le site officiel, le projet du compilateur ST 200 est cité [13] en tant qu'utilisateur de Valgrind.

J'ai aussi appris à utiliser des outils que je ne manipulais pas :

GDB : Il s'agit d'un débogueur [14], il permet de voir à l'intérieur d'un programme en exécution ce qu'il se passe.

Indent : Ce programme [15] permet d'indenter automatiquement à son goût les sources d'un programme.

Général

Je pense que suite à ce stage je ne développerai plus comme avant. En effet, mon maître de stage m'a appris à bien étudier le contexte dans lequel on est amené à programmer, et à décortiquer la moindre chose.

Finalement, j'ai pu me perfectionner dans la lecture de l'anglais, une langue incontournable dans le domaine scientifique !

[10] http ://www.doxygen.org
[11] http ://www.cvshome.org
[12] http ://valgrind.kde.org
[13] http ://valgrind.kde.org/users.html
[14] http ://www.gnu.org/software/gdb/gdb.html
[15] http ://www.gnu.org/directory/GNU/indent.html

1.4.5 Observations

J'ai pu observé durant ce stage les méthodes de travail des chercheurs. C'est ainsi que je me suis rendu compte que beaucoup d'idées étaient échangées sur les tableaux et pendant les réunions de groupe de travail.

J'ai pu voir qu'il est nécessaire d'avoir une grande maîtrise de son domaine, un esprit critique et une certaine rigueur.

Finalement ce stage m'a permis de me surpasser, grâce à ma motivation initiale, à l'environnement de développement du LAO 2 et à l'encadrement de mon maître de stage. En effet, j'ai dû assimiler un nouveau langage de programmation et de nouvelles notions en très peu de temps, et nous avons finalement obtenu un résultat impressionnant, comme nous le verrons à la section 2.6.2.

Je n'ai qu'un seul regret : la durée du stage, trois mois, c'est trop court pour se mettre à réfléchir par *soi-même* sur un sujet de recherche…

Chapitre 2

Travail effectué

2.1 L'environnement de développement

2.1.1 Qu'est-ce qu'un compilateur ?

Un compilateur [14] est un programme qui prend en entrée un programme écrit dans un langage et produit en sortie un autre programme écrit dans un autre langage.

Le langage du programme source est appelé *langage source*, il peut être : Fortran, C, Java ... Celui du programme cible est appelé *langage cible*, cela peut être de l'assembleur, du code machine, ou un autre langage...

Le compilateur doit fondamentalement :
- conserver le sens du programme ;
- améliorer le code [1].

Les propriétés importantes d'un compilateur sont :
- code produit efficace (rapidité, mémoire) ;
- informations retournées en cas d'erreurs, deboguage ;
- rapidité de compilation.

Dans notre cas, le dernier point importe peu puisque nous souhaitons avant tout créer du code rapide et peu encombrant pour des systèmes enfouis, nous pouvons donc utiliser des algorithmes complexes et longs pour atteindre cet objectif.

L'interface

La première tâche du compilateur est de comprendre l'expression du langage source. Cela se fait en plusieurs étapes :
- l'analyse syntaxique permet de vérifier que l'expression est bien une phrase du langage source syntaxiquement correcte ;
- l'analyse sémantique permet de vérifier que l'expression a un sens dans le langage source. Par exemple, en Français, « La montagne roule une blague. » est syntaxiquement correcte, mais pas sémantiquement.

Cette tâche est réalisée par ce que l'on appelle le « front-end ».

Améliorations

Le compilateur peut très souvent tirer parti du contexte dans lequel se trouve une expression pour optimiser son implémentation. Cette phase de la compilation est généralement

[1] il est possible d'améliorer la vitesse d'exécution, ou la taille de mémoire utilisée.

séparée en des passes d'*analyse* et des passes de *transformations*. Les analyses vont permettre de mettre en place les objets utilisés lors des transformations.

Cette phase est parfois appelée « middle-end ».

Production de code

C'est la partie qui diffère avec les interpréteurs. Le compilateur parcourt la structure de données qui représente le code et émet un code équivalent dans le langage cible. Il doit choisir les instructions à utiliser pour implémenter chaque opération (sélection d'instruction), décider quand et où copier les valeurs entre les registres et la mémoire (allocation de registres), choisir un ordre d'exécution pour les instructions choisies (ordonnancement d'instructions). Souvent les optimisations de ces différentes phases entre en conflit.

Cette phase est aussi appelée « back-end ». Nous avons finalement la chaîne de compilation représentée à la figure 2.1.

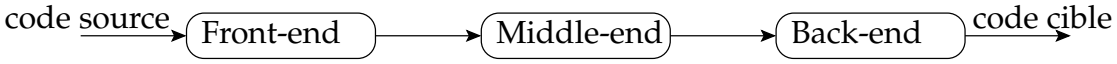


FIG. 2.1 – Chaîne de compilation

2.1.2 L'Open64 et le LAO 2

Nous avons implémenté l'allocateur de registres dans le LAO 2 qui est un programme *branché* dans l'Open64. L'Open64 est en fait un ensemble d'outils de développement de compilateur sous licence GNU GPL, successeur du compilateur Pro64 de SGI pour l'Itanium d'Intel.

L'équipe MCDT de STMicroElectronics en a modifié une partie [2] dans le but de générer du code pour leur processeur ST 200. Celui-ci fait partie de la famille des processeurs VLIW (Very Long Instruction Word), il peut exécuter des *paquets d'instructions* et donc faire du *vrai* parallélisme. Ils ont aussi créé le fameux *module* LAO 2 pour l'Open64 capable de prendre en compte les caractéristiques du ST 200 pour produire un code optimisé.

2.1.3 Le langage XCC

Afin que les sources du projet LAO 2 soient uniformisées et pour simplifier la programmation, Benoît Dupont de Dinechin a créé une extension du C, nommée XCC (par analogie avec l'extension des fichiers .cxx du C++). Celui-ci apporte au C quelques notions de la programmation orienté objet :

- dérivation ;
- constructeurs et destructeurs ;
- accesseurs et mutateurs ;
- relations *ami*, *privé*, *public* ;
- ...

Prenons par exemple le code suivant, définissant la structure « ensemble de pointeurs » à la figure 2.2

La première ligne précise que l'on déclare la structure *amie* pour le module, la ligne contenant $//@args$ déclare les paramètres du constructeur et les lignes suivantes permettent d'hériter de la structure *IStack_*, désignant une pile.

[2] entre autres ce que l'on appelle le « back-end ».

```
//@XCC._h
struct PtrSet_ {
  //@args Memory memory, int32_t maxCount
  IStack_ ISTACK_;
    //@access ISTACK PtrSet__ISTACK_(this)
  //@ctor IStackCTOR(PtrSet_ISTACK(this),
                     memory, maxCount,
                     sizeof(PtrSetMember));
  //@dtor IStackDTOR(PtrSet_ISTACK(this));
    //@access MEMORY IStack_MEMORY(PtrSet_ISTACK(this))
    //@access MAXCOUNT IStack_MAXCOUNT(PtrSet_ISTACK(this))
    //@mutate MAXCOUNT IStack__MAXCOUNT(PtrSet_ISTACK(this))
[...]
};
```

FIG. 2.2 – Exemple de structure en XCC

Une fois que l'on a écrit un fichier en XCC, par exemple *mod.xcc*, on le compile grâce à l'utilitaire *xccgen* qui nous fournit en sortie quatre fichiers écrit en langage C :

- *mod._h* contient les déclarations des membres amis ainsi que les définitions des structures complétées par *xccgen* ;
- *mod.h* contient les déclarations des membres *publics* ;
- *mod._c* contient les définitions des fonctions *inline* ;
- *mod.c* contient les déclarations des membres privés ainsi que les définitions de tous les autres membres.

Si l'on veut être *ami* du module, il suffit d'inclure *mod._h*.

2.1.4 La bibliothèque CCL

La bibliothèque CCL est écrite en XCC, par Benoît Dupont de Dinechin et Christophe Guillon. Elle est formée de toutes sortes de conteneurs :

- listes ;
- ensembles ;
- graphes orientés ;
- ...

sur lesquels on peut effectuer des opérations standards :

- itérations ;
- insertions ;
- suppressions ;
- ...

Cela permet de simplifier la programmation du LAO 2 puisque l'on n'a plus à se soucier des structures de données, sachant que l'on trouve pratiquement tout dans cette bibliothèque. Il manque tout de même une structure de graphe non-orienté utilisée dans l'algorithme que nous avons implémenté, nous l'avons donc développé nous-même !

Reprenons l'exemple de la structure « ensemble de pointeurs » et observons sur le code de la figure 2.3 quelques opérations que l'on peut effectuer.

```
// voici une itération
PtrSet_FOREACH(set, pointeur) {
        fprintf("%p\n", pointeur);
} PtrSet_ENDEACH;

// voici une insertion
PtrSet_insert(set, &i);

// voici une suppression
PtrSet_remove(set, &j);
```

FIG. 2.3 – Exemple d'opération sur une structure CCL

2.2 Problématique

2.2.1 Introduction

Le nombre de variables est une des différences fondamentales entre les langages de haut niveau et les langages machine. Un langage de haut niveau peut utiliser un nombre infini de variables, tandis qu'un langage machine en possède un nombre fini, appelées *registres*. Le compilateur doit *assigner* les variables aux registres, tout en assurant la validité de leurs valeurs.

Ainsi, il se peut qu'à certains endroits du programme, il ne puisse plus avoir suffisamment de registres libres pour contenir toutes les variables *en vie*. Lorsque cela arrive, le compilateur doit *vider* certaines d'entre elles en mémoire, pour libérer de la place. Malheureusement, l'utilisation ou la définition d'une variable est bien plus coûteuse lorsqu'elle se situe en mémoire, il faut donc allouer au mieux les registres pour minimiser le temps d'exécution ainsi que la taille de mémoire dynamique utilisée.

2.2.2 Approche

L'allocation de registres est un problème complexe, il faut :
- affecter un registre à chaque variable, en assurant l'intégrité de leurs valeurs ;
- optimiser les cas où l'on est obligé de mettre une variable en mémoire ;
- supprimer les opérations de copie en essayant d'assigner le même registre à deux variables différentes.

Dans notre cas nous *approchons* le problème d'allocation de registre par un problème de coloration de graphe. Chaque noeud du graphe représente une variable, et deux noeuds sont connectés par un arc si les variables qu'ils représentent interfèrent entre elles, c'est-à-dire si elles sont en vie en même temps à un endroit du programme. C'est sur dernier point que l'on perd beaucoup d'information, en effet nous ne savons pas à quel moment deux variables interfèrent.

Le problème de coloration de graphe est tout de même NP-Complet [3] dans le cas d'un graphe *quelconque*. Même si dans notre cas nous ne sommes pas trop préoccupés par le temps de compilation, un algorithme exponentiel n'est pas envisageable, il faut donc utiliser des *heuristiques*. Aujourd'hui, la plupart des compilateurs se base sur cette approche.

[3] la *classe P* est la classe des problèmes résolvables en temps polynomial, la *classe NP* est la classe des problèmes dont savoir si la solution est correcte ou non est résolvable en temps polynomial, et la *classe NP-complet* est la classe des problèmes de la *classe NP* dont si on sait les résoudre en temps polynomial alors *P=NP*. Notre problème fait donc parti des problèmes de la *classe NP* les *plus difficiles*.

Il faut noter qu'il existe une classe de noeuds particuliers représentant les registres. En effet, les registres peuvent être directement utilisés comme variables dans un programme, entre autre pour le passage de paramètres. Le graphe d'interférence possède donc des noeuds *précolorés* représentant les registres.

2.3 L'algorithme « Iterated Register Coalescing »

L'algorithme « Iterated Register Coalescing » [9] est un raffinement de l'algorithme d'allocation de registres via la coloration de graphe [10]. Il prend en entrée un graphe de flot de contrôle, c'est-à-dire une structure interne au compilateur désignant une procédure, où chaque noeud représente un bloc de base, c'est-à-dire un morceau de code ne contenant aucun branchement sauf au début et à la fin. Les arcs représentent un saut dans le flot de contrôle. Il y a deux blocs de base spéciaux :

– le bloc d'entrée, par lequel le flot de contrôle entre ;
– le bloc de sortie, par lequel le flot de contrôle sort.

Pour illustrer ce qu'est un graphe de flot de contrôle, prenons un bout de code C que nous traduisons en instructions à la figure 2.4, où $r_{[1-3]}$ représente un registre.

exemple de code C :

```
int f(int a, int b) {
        int d=0;
        int e=a;

        do {
                d = d + b;
                e = e - 1;
        } while(e > 0);

        return d;
}
```

traduit en instructions :

```
enter :
        c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e − 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return
```

FIG. 2.4 –

Il existe deux classes de registres :

– les registres « caller-save », qui doivent être sauvegardés avant un appel à une fonction, dans notre exemple c'est le cas de r_1 et r_2 ;
– les registres « callee-save », qui doivent être sauvegardés dans la fonction appelée, dans notre exemple c'est le cas de r_3.

Nous extrayons ensuite le graphe de flot de contrôle à la figure 2.5.

2.3.1 Graphe d'interférences

Dans chacun des blocs de base du code, nous calculons les plages de vie des variables, cela est réalisé à partir de la fin jusqu'au début, de manière linéaire puisqu'il n'y a pas de saut. Reprenons le premier graphe de flot de contrôle et effectuons le calcul des plages de vie à la figure 2.6, où un carré noir représente une définition et un hexagone blanc une utilisation.

L'analyse des plages de vie des variables permet, en plus, de connaître la « pression registre », c'est-à-dire le nombre maximal de variables en vie au même instant. La *pression registre*

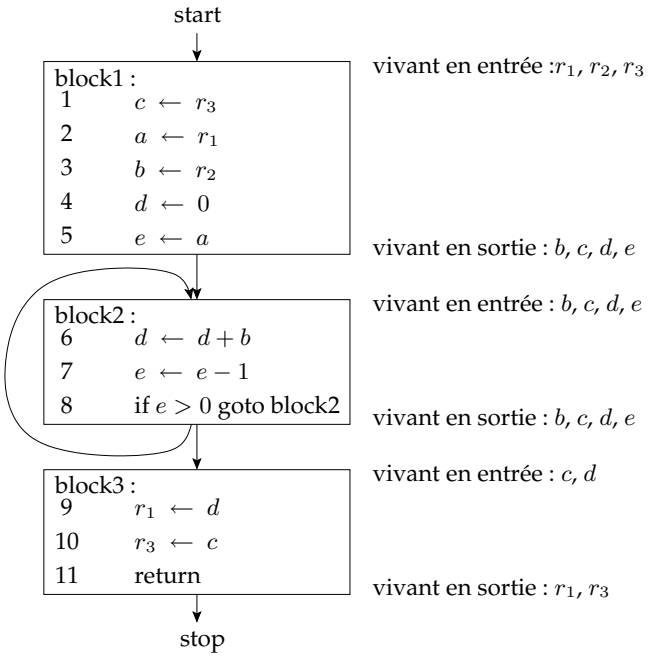


FIG. 2.5 – Graphe de flot de contrôle

détermine « MaxLive », le nombre de registres minimum nécessaire à l'allocation sans utiliser la mémoire. Sur la figure 2.6 nous comptons le nombre de traits verticaux (les plages de vie) entre chaque ligne (les dates). La pression registre maximale étant de 4 entre les dates 4 et 8, *MaxLive* est alors égal à 4.

À partir du calcul des plages de vie de la figure 2.6, nous pouvons calculer le graphe d'interférences à la figure 2.7.

En fait, lorsque nous construisons ce graphe, nous rajoutons des arcs en pointillés représentant des instructions MOVEs, celle-ci sont simplement des copies de variables. La présence de ces arcs MOVEs sera justifiée à la section 2.3.5.

2.3.2 Simplification

Le coloriage du graphe d'interférences se fait premièrement par l'heuristique de Brooks [6], très simple et très efficace.

Supposons qu'il existe un noeud possédant *strictement* moins de K voisins, où K est le nombre de registres disponibles. Dans le *pire des cas*, tous ses voisins ont une couleur différente, donc il reste *au moins* une couleur disponible pour lui. Ce noeud peut donc être enlevé du graphe car la colorabilité de ce dernier ne dépend pas de lui. En conséquence, tous ses voisins ont perdu un degré, et sont peut-être passés d'un degré élevé [4] à un degré faible [5].

En fait, lorsque l'on dit que l'on enlève le noeud du graphe, on le place dans une *pile*. Ainsi, lorsque tous les noeuds sont supprimés du graphe (sauf les noeuds précolorés) et placés dans cette *pile de coloration*, on dépile cette dernière pour affecter les couleurs aux noeuds, c'est-à-dire assigner les registres aux variables.

Comme nous allons le voir à la section 2.3.5, nous n'appliquons pas cette phase aux noeuds qui sont impliqués dans un arc MOVE.

[4] supérieur ou égal à K

[5] strictement inférieur à K

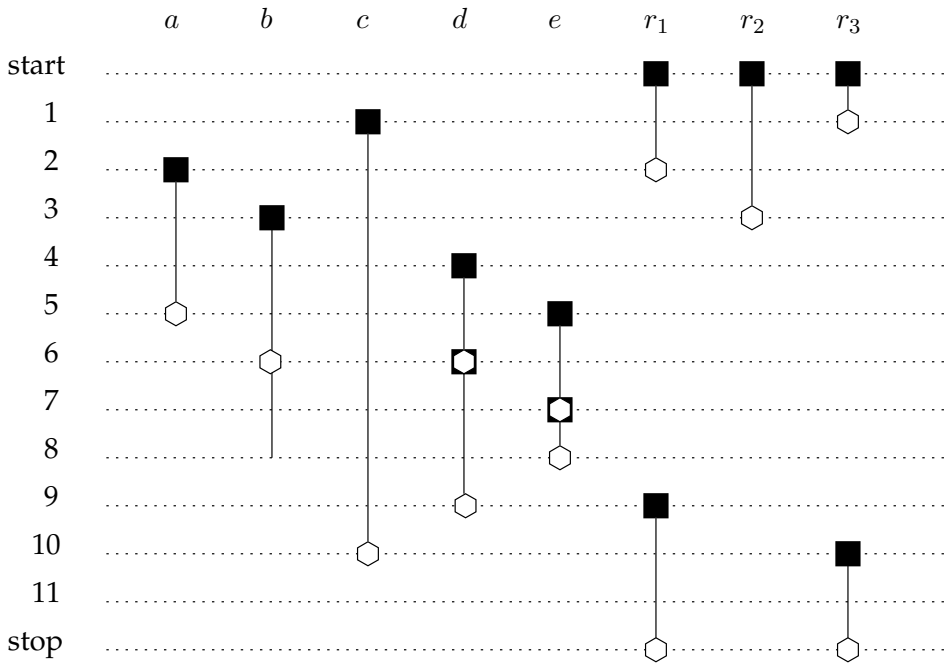


FIG. 2.6 – Plages de vie

2.3.3 Vidage en mémoire, « spill »

S'il n'y a plus de noeuds de faible degré, nous choisissons un noeud de degré élevé pour être *potentiellement vidé* en mémoire. Ce noeud est supprimé du graphe et placé dans la *pile de coloration*, dans l'espoir qu'il sera tout de même coloriable comme nous le verrons à la section 2.3.4.

Prenons le cas de la figure 2.8 et d'une machine à 2 registres, tous ces noeuds sont de *degré élevé*. Il n'y a aucune *simplification* possible, donc nous *vidons en mémoire* le noeud a. Tous les autres noeuds peuvent ensuite être enlevés du graphe grâce à la phase de simplification : d'abord b, puis d, et finalement c. Comme nous le verrons à la section 2.3.4 nous serons tout de même capable de colorier le noeud a...

Certaines variables sont plus coûteuses que d'autres à *vider en mémoire*. Nous calculons le coût du vidage en mémoire selon la formule :

$$\frac{\sum_{i=1}^{n} \text{load\_cost} * \text{freq}_i + \sum_{j=1}^{m} \text{store\_cost} * \text{freq}_j}{\text{degree}}$$

où n est le nombre de définitions de la variable, m son nombre d'utilisations, les valeurs *load_cost* et *store_cost* sont respectivement les *coûts* de chargement et de stockage en mémoire, $freq_x$ est une estimation de la fréquence du bloc de base numéro x, et *degree* est le degré du noeud représentant la variable.

Cette formule permet de déterminer quelles sont les variables qui sont peu utilisées et *a fortiori* rarement dans les boucles. En effet, il serait catastrophique de vider en mémoire une variable utilisée dans une boucle, puisqu'il faudrait la charger et la stocker à chaque itération. On divise par le degré afin de pouvoir supprimer du graphe les noeuds qui ont le plus grand nombre de voisins, et donc diminuer le degré d'un maximum de noeuds.

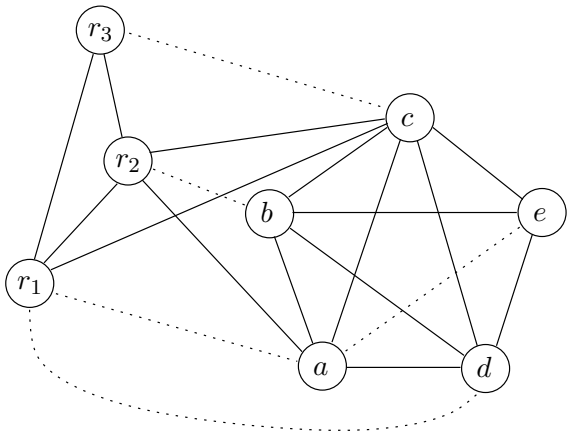


FIG. 2.7 – Graphe d'interférences

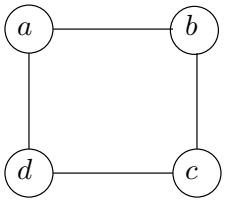


FIG. 2.8 – Graphe d'interférences

2.3.4 Coloration

Nous dépilons la *pile de coloration*, en assignant des couleurs aux noeuds. Dans cette phase lorsque l'on colorie un noeud, nous prenons soin de vérifier la couleur de ses voisins, et de lui assigner une couleur différente.

En ce qui concerne les noeuds de la pile qui sont *vidés en mémoire*, nous essayons tout de même de les colorier comme précédemment, et en cas d'échec, nous les vidons *effectivement* en mémoire.

Reprenons l'exemple de la figure 2.8 et regardons ce qu'il se passe lorsque tous les noeuds du graphe d'interférences ont été placés dans la pile de coloration comme ci-dessous :

c	$\leftarrow$ variable simplifiée
d	$\leftarrow$ variable simplifiée
b	$\leftarrow$ variable simplifiée
a	$\leftarrow$ variable vidée en mémoire

On dépile en affectant des registres aux variables comme cela :

$$c \Leftarrow r_1$$
$$d \Leftarrow r_2$$
$$b \Leftarrow r_2$$
$$a \Leftarrow r_1$$

Nous avons donc gagné un vidage en mémoire, puisque a est finalement coloriable ! Lorsque l'on place une variable dans la pile de coloration parce qu'il faut la vider en mémoire, nous parlons de « vidage potentiel en mémoire ». Lorsque cette variable est effectivement vidée en mémoire lors de la phase de coloration, nous parlons de « vidage effectif en mémoire ».

Comme nous l'avons vu à la section 2.2.2 il existe une perte d'information lorsque l'on utilise ce principe de graphe d'interférences. En effet, s'il l'on est amené à vider en mémoire une variable parce qu'il existe une zone du code où la *pression registre* est trop importance alors il faudrait recharger et stocker cette variable *partout* dans le code, puisque l'on ne sait pas d'où vient exactement le débordement de capacité.

Lorsque l'on a effectivement un vidage en mémoire, il faut relancer l'algorithme complet, c'est-à-dire à partir de la création du graphe d'interférences. En effet, nous avons changé le code puisque nous avons inséré des opérations d'accès à la mémoire afin de réduire la *pression registre*, et par conséquence le graphe d'interférences a changé. Comme précédemment, le fait que l'on ait une perte d'information relative aux dates nous empêche de faire une simple mise à jour du graphe d'interférences.

Jusqu'à présent nous avons vu l'algorithme de base de l'allocation de registres par coloration de graphe, la prochaine étape présente le raffinement apporté par l'*Iterated Register Coalescing* permettant de traiter le problème de suppression des opérations de copie.

2.3.5 Fusion

Lorsque deux variables sont liées par une instruction MOVE et qu'elles n'interfèrent pas entre elles, alors ces variables peuvent être fusionnées. Ainsi, au lieu d'avoir au pire deux registres (couleurs) d'utilisés pour assigner (colorier) ces deux variables (noeuds), nous n'en avons besoin que d'un(e) seul(e). De plus, la fusion de noeuds permet de supprimer complètement l'instruction MOVE en question et donc gagner du temps d'exécution.

Il faut tout de même faire attention à ce que le graphe obtenu par cette opération ne soit pas plus difficile à colorier, pour cela nous utilisons deux heuristiques *conservatives*, c'est-à-dire que si le graphe était K-Brooks-coloriable [6] *avant* la fusion, alors il l'est *après* :

heuristique 1 : Celle-ci est basée sur l'heuristique de Briggs [4]. Les noeuds a et b peuvent être fusionnés si le nombre des voisins de degré élevé de a et de b est inférieur à K.

heuristique 2 : Celle-ci est basée sur l'heuristique de George [9]. Le noeud a peut être fusionné avec le noeud précoloré b si tous les voisins de degré élevé de a sont aussi voisin de b. Nous l'utilisons donc pour fusionner un registre (noeud précoloré) avec une variable.

Il existe une autre classe, celle des MOVEs contraints, c'est-à-dire que l'on ne peut pas fusionner, que cela soit conservatif ou non. En effet, nous ne pouvons pas fusionner deux noeuds qui interfèrent.

2.3.6 Gel

Si ni la phase de *simplification*, ni la phase de *fusion* ne peuvent être appliquées, nous cherchons un noeud de degré faible impliqué dans un arc MOVE. Nous *supprimons* alors les arcs MOVEs dans lesquels ce noeud est impliqué, c'est-à-dire que nous abandonnons tout espoir de supprimer les instructions MOVEs correspondantes. Cela a pour conséquence que ce noeud, ainsi que ceux relatifs au MOVE supprimé, peuvent maintenant être considérés dans la phase de *simplification*.

2.3.7 Organisation

Au final, toutes ces phases ce succèdent selon la figure 2.9.

[6] coloriable avec K couleurs en utilisant l'algorithme de Brooks.

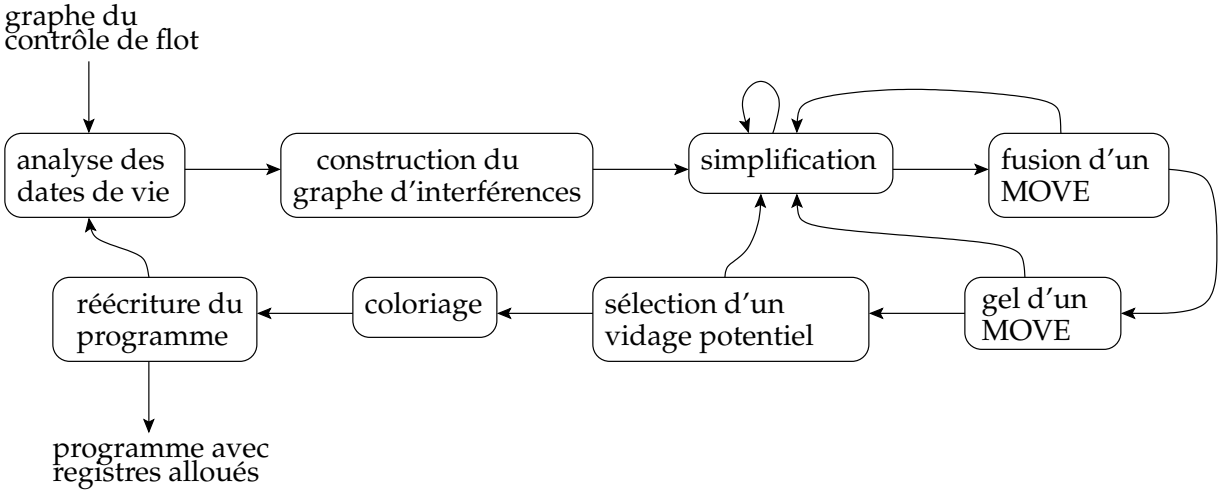


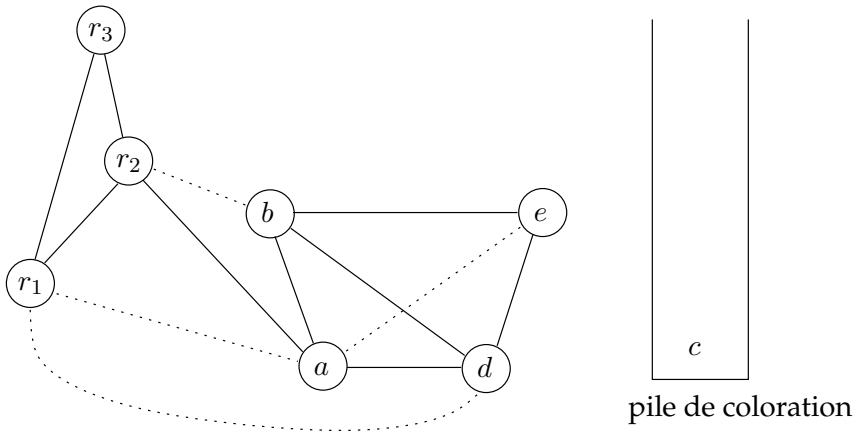
FIG. 2.9 – Phases de l'allocateur de registres

2.4 Exemple concret

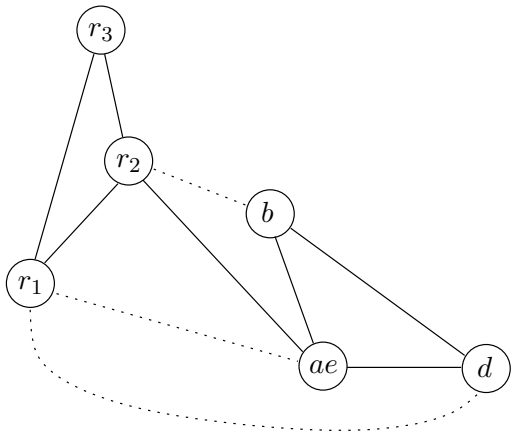
Reprenons l'exemple de code C à la section 2.3 pour un processeur à 3 registres, nous avions déjà calculé le graphe d'interférence correspondant, disponible à la figure 2.7.

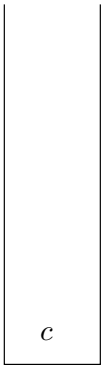
Dans ce graphe il n'y a aucune opportunité de *simplifier* ou de *geler* (car tout les noeuds non précolorés ont un degré ≥ 3), n'importe quelle tentative de *fusion* produirait un noeud résultant avec au moins 3 voisins, nous devons alors *vider en mémoire* une variable. Nous calculons donc le coût de vidage en mémoire...

Le noeud c a le plus petit coût, car il interfère avec beaucoup de variables mais n'est pas souvent utilisé, nous pouvons donc le *vider* en premier. Nous obtenons alors le graphe de la figure suivante :



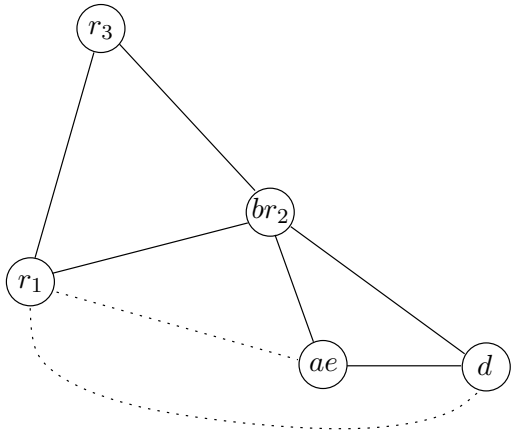
Nous ne pouvons toujours pas *simplifier* de noeuds car les noeuds de faible degré sont impliqués dans des arcs MOVEs, en revanche nous pouvons *fusionner* a et e, puisque le noeud résultant aura moins de 3 voisins de degré élevé.

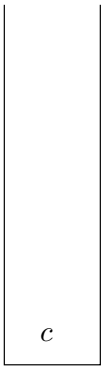




pile de coloration

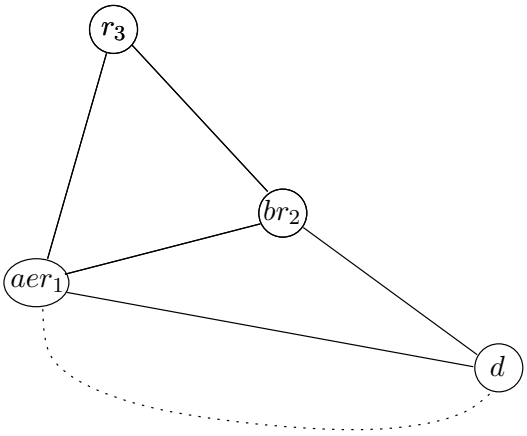
Pour les même raisons que précédemment, nous pouvons *fusionner* ae et r_1, ou b et r_2. Effectuons ce dernier cas.

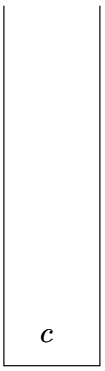




pile de coloration

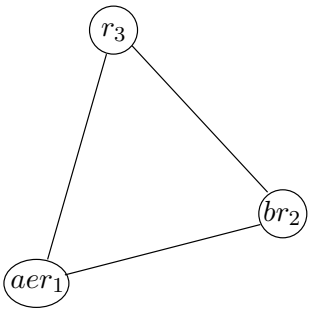
De même nous pouvons *fusionner* ae et r_1, ou d et r_1. Effectuons le premier cas.

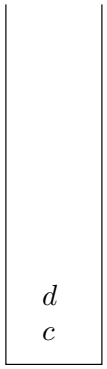




pile de coloration

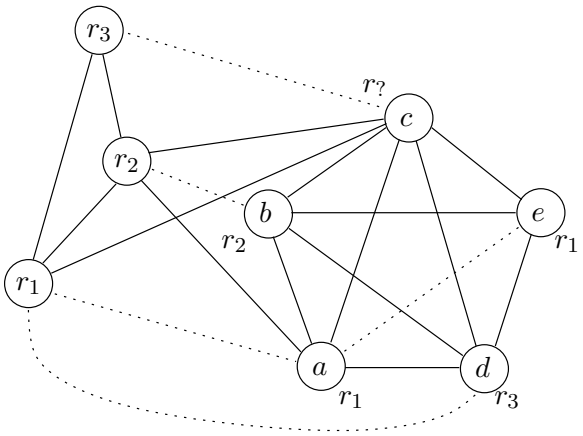
Dans un premier temps, nous ne pouvons toujours pas *simplifier* de noeuds, mais nous ne pouvons pas non plus *fusionner* aer_1 et d car l'arc MOVE est contraint, en effet les noeuds aer_1 et d interfèrent. Nous gèlons donc cet arc MOVE, ce qui permet finalement de *simplifier* d.





pile de coloration

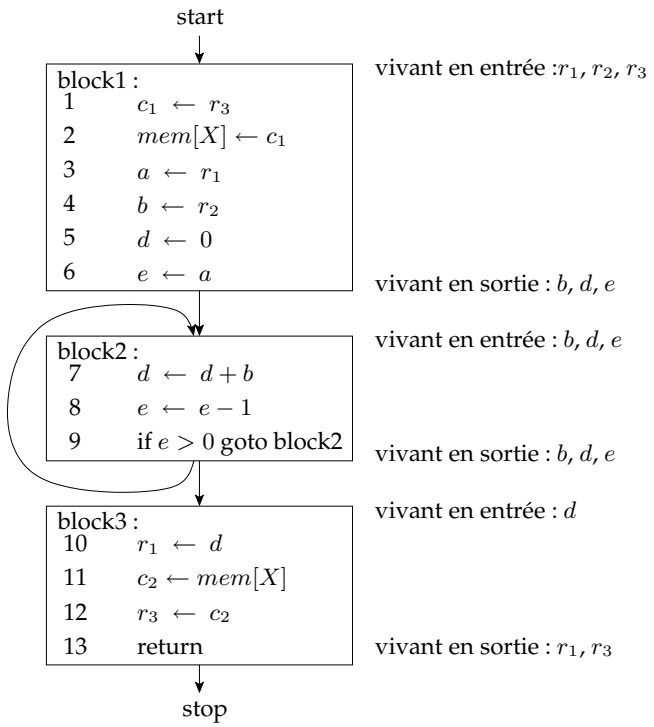
Nous obtenons un graphe avec uniquement des noeuds précolorés, nous dépilons donc les noeuds (variables) de la pile et nous leurs assignons une couleur (registre). Tout d'abord, nous prenons d qui peut être assigné à la couleur de r_3. Les noeuds a, b, e ont déjà été assignés par la *fusion*. Mais le noeud c, qui était *potentiellement vidé* se transforme en un noeud *effectivement vidé*, puisqu'il n'y a plus de couleur pour lui.



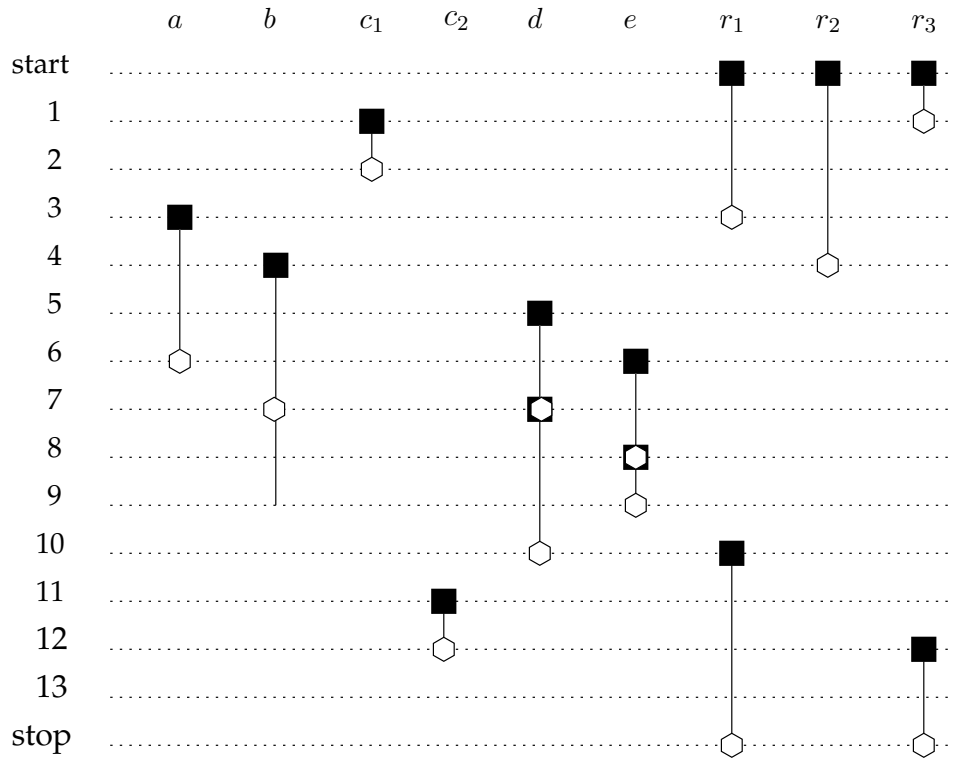
Puisqu'il y a *effectivement* un vidage en mémoire, nous devons ré-écrire le programme, en y rajoutant les opérations de sauvegarde et de chargement. Pour chaque utilisation (définition) de c, nous créons une nouvelle variable, et la chargeons (sauvegardons) immédiatement après (avant).

```
enter :
        c1 ← r3
        mem[X] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop :  d ← d + b
        e ← e − 1
        if e > 0 goto loop
        r1 ← d
        c2 ← mem[X]
        r3 ← c2
        return
```

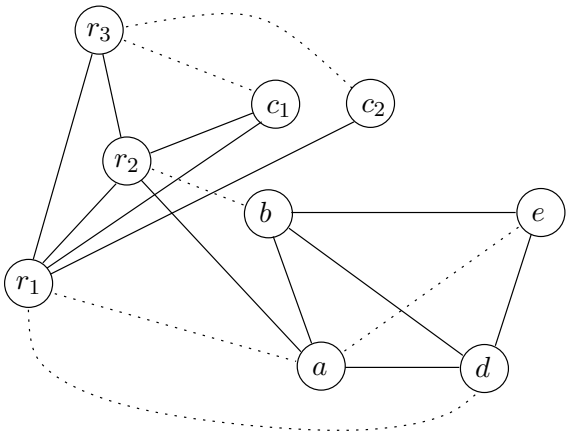
À partir de ces instructions, nous calculons le graphe de flot du contrôle suivant.



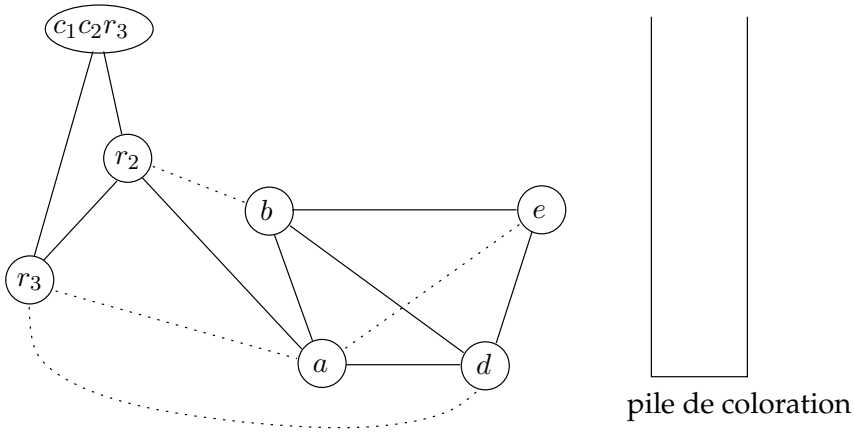
Puis nous en extrayons les plages de vie des variables.



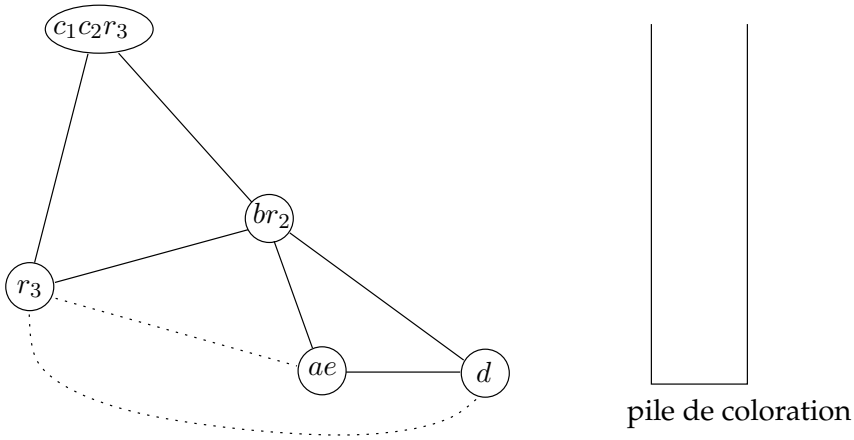
Enfin nous reconstruisons le graphe d'interférence.



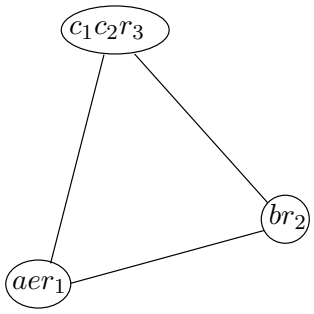
Nous pouvons immédiatement *fusionner* c_1 et r_3, puis c_2 et $c_1 r_3$.

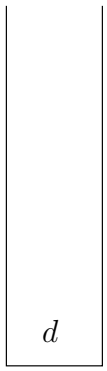


Puis, comme précédemment, nous pouvons *fusionner* a et e, ainsi que b et r_2.



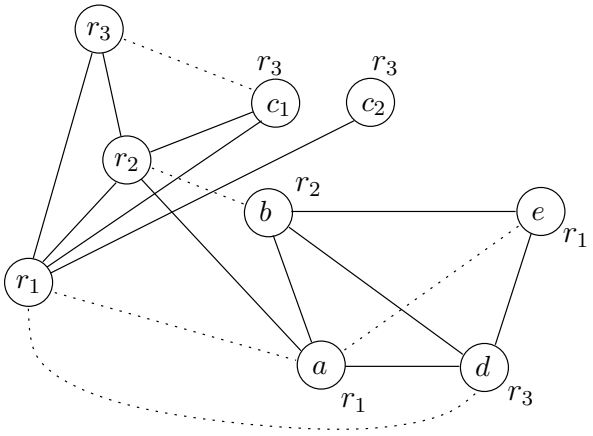
Comme précédemment, nous pouvons *fusionner* ae et r_1, puis *simplifier* d.





pile de coloration

Nous arrivons donc à la dernière phase, où nous vidons la *pile de coloration*. Il n'y a qu'un seul noeud, tous les autres sont *fusionnés* ou *précolorés*. Nous sommes obligés d'assigner à d la couleur du registre r_3.



Nous pouvons maintenant ré-écrire le programme en utilisant cet assignement.

```
enter :
        r3 ← r3
        mem[X] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop :  r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← mem[X]
        r3 ← r3
        return
```

Finalement nous supprimons toutes les instructions MOVEs dont la destination et la source sont identiques. Il est possible que des MOVEs que nous n'avons pas fusionnés disparaissent, notamment lorsque *par hasard* on a réussi à assigner le même registre pour la source et pour la destination.

```
enter :
        mem[X] ← r3
        r3 ← 0
loop :  r3 ← r3 + r2
        r1 ← r1 − 1
        if r1 > 0 goto loop
        r1 ← r3
        r3 ← mem[X]
        return
```

Le programme possède finalement une seule instruction MOVE non supprimée.

2.5 Spécifications du ST 200

Bien évidemment l'algorithme décrit ci-dessus nécessite d'être adapté aux spécificités de l'architecture ST 200 [13].

2.5.1 Architecture RISC

Le processeur ST 200 est une architectures RISC, c'est-à-dire qu'il possède un nombre très limité d'instructions et que l'on ne peut pas effectuer d'opérations avec une variable en mémoire. Il faut donc avoir un registre de libre pour y placer la valeur située en mémoire.

Si l'on pouvait effectuer les opérations directement avec une variable en mémoire, la phase de vidage serait complètement différente et il ne serait pas utile de relancer l'algorithme complet puisqu'il n'y aurait pas besoin de libérer un registre pour contenir la valeur en mémoire ! Comme on vient de le voir, ce n'est malheureusement pas le cas...

2.5.2 Classe de registres

Il existe plusieurs classe de registres dans le processeurs ST 200 :
- 8 registres booléens, utilisés entre autres pour les opérations de comparaisons ;
- 64 registres généraux, utilisés entre autres pour les opérations sur les valeurs entières [7].

Nous avons donc *partitionné* les registres en deux et alloué de manière indépendante les variables booléennes et entières.

2.5.3 Registres dédiés

Il existe des registres *dédiés* auxquels il n'est pas possible d'assigner une variable, ces registres sont :
- le registre « wired » R0, qui contient toujours la valeur 0 ;
- le registre de pile R12, qui contient l'adresse de la pile du programme ;
- le registre de retour R63, qui contient l'adresse de branchement à effectuer après la fin de la fonction.

[7] les opérations sur les flottants sont émulées

2.5.4 Les opérations « clobbers »

Certaines opérations, dites « clobbers » modifient implicitement les valeurs de certains registres. C'est le cas par exemple avec l'instruction CALL [8] qui n'assure pas l'intégrité des valeurs stockées dans les registres booléens ainsi que dans la majorité des registres généraux [9].

2.6 Résultats

2.6.1 Les différents allocateurs

Il existe 4 allocateurs de registres pour le LAO 2.
- l'allocateur de registres du compilateur Open64 [10] ;
- l'allocateur « Localize » [11], développé par l'équipe MCDT de ST pour valider l'intégration d'un allocateur à la place de celui de l'Open64 (environ 500 lignes de code XCC) ;
- l'allocateur « Linear Scan » [12] , implémenté par Florent Blachot, dans le cadre de son stage de DEA [3], utilisé dans les compilateurs *just in time* [13] (environ 1300 lignes de code XCC) ;
- l'allocateur « Iterated Register Coalescing » implémenté dans le cadre de mon stage de Maîtrise informatique (environ 3300 lignes de code XCC et commentaires Doxygen).

Le but de cette implémentation est de fournir à mon maître de stage une base pour qu'il puisse ouvrir de nouvelles voies de recherche en optimisation et compilation, et pour qu'il puisse comparer ses résultats avec une technique performante. L'équipe *MCDT* de STMicroElectronics fut très satisfaite de ce travail, car c'était la première fois qu'un développement d'une telle envergure s'effectuait hors de leur entreprise, cela leur a permis de valider leurs outils de développement de manière *externe*. Ils ont finalement intégré cet allocateur de registre dans leur compilateur...

2.6.2 Tests de performances

La figure 2.10 présente des tests de performances de notre implémentation dans le LAO 2, où l'on a compilé quelques programmes avec les différents allocateurs que l'on a ensuite exécuté sur un simulateur à cache parfait[14]. Pour comparer les résultats, le simulateur compte le nombre de cycles des programmes pour chacunes des versions compilées avec les différents allocateurs, les chiffres dans les colonnes du tableaux de la figure 2.10 représentent le gain par rapport à l'allocateur de l'open64.

En moyenne, notre implémentation de l'Iterated Register Coalescing est plus performant de 10% que le LinearScan ! En revanche, elle est moins performante de 2% à 15% que l'allocateur de l'Open64... Nous savons exactement d'où cela provient et nous savons aussi comment avoir finalement de meilleurs résultats, comme nous le verrons aux sections 2.7.2 et 2.7.3.

[8] c'est un appel de fonction.

[9] plus spécifiquement tous les registres sauf R7 à R12 et R0.

[10] le principe de cet allocateur est de mélanger l'Iterated Register Coalescing et une technique de détection de haute pression registre en vidant en mémoire les variables globales si besoin.

[11] le principe de cet allocateur est de vider en mémoire les variables globales lorsqu'il est nécessaire, ainsi l'allocation se fait de manière *locale*.

[12] le principe de cet allocateur est d'allouer les registres de façon gloutonne, c'est-à-dire linéaire.

[13] qui compile juste avant l'exécution.

[14] *toutes* les instructions et données sont *toujours* dans le cache

	Rapport de performances (en %) entre l'Open64 et	
Programmes	le Linear Scan	l'Iterated Register Coalescing
Tri par tas	100	100
Tri fusion	70	95
Tri rapide	80	98
Cryptage MD5	92	100
Cryptage RSA 2	96	96
Encodage DivX d'un DVD	86	98
Décodage Mpeg4	84	97

FIG. 2.10 – Tests de performances

2.7 Projet de TER

Comme je l'avais dit à la section 1.4.5, je trouve dommage que l'on n'ait pas eu suffisamment de temps durant ce stage pour réfléchir par soi-même. C'est pour cela que je souhaite poursuivre ce projet dans le cadre de mon Travail d'Étude et de Recherche de Maîtrise.

2.7.1 Accélération du temps de compilation

Cela n'est pas une priorité car le code produit n'est pas plus rapide, seul le temps de compilation est accéléré. Nous avons déjà effectué quelques *optimisations* dans le même style, inspirées de l'implémentation de l'*Iterated Register Coalescing* dans un compilateur ML pour architecture RISC [11].

Ignorer la phase de fusion

Toute la phase de *fusion conservative* peut être ignorée lorsqu'il y a *effectivement* un vidage en mémoire, car le code sera modifié par l'insertion de nouvelles instructions de chargement et stockage en mémoire. Techniquement, nous pouvons *parfois* savoir s'il y aura *effectivement* un vidage en mémoire pendant la construction du graphe d'interférence, il suffit de calculer *MaxLive*. Dans ce cas là, il n'est pas nécessaire d'effectuer la phase de *fusion* lors de la première passe de coloriage.

Minimisation de l'allocation dynamique

L'allocation dynamique de mémoire est l'une des opérations les plus gourmandes en temps, nous avons déjà travaillé sur sa minimisation mais il reste encore quelques améliorations à effectuer.

2.7.2 Raffinement de l'utilisation des registres

Nous avons vu à la section 2.5.3 qu'il existait trois registres dédiés, parmi eux le registre d'adresse de retour R63. Celui-ci n'est utilisé qu'à la fin d'une fonction pour pouvoir revenir au point d'appel, par précaution nous considérions comme non-assignable. En fait il peut être assigné sous certaines conditions [15], cela nous fait donc un registre en plus pour les passages à grosse *pression registres* !

[15] certaines instructions, telle que la multiplication, ne peuvent pas le définir.

Nous avions vu à la section 2.5.2 qu'il y avait deux classes de registres, une pour les booléens et une pour les entiers. Il est techniquement possible de stocker une valeur booléenne dans un registre entier grâce à des instructions spécifiques. Nous pouvons donc *vider* les valeur booléennes dans les registres généraux en cas de grosse *pression registres booléenne*. Cela est fort intéressant car un accès entre registres de classes différentes est bien plus rapide qu'un accès à la mémoire !

2.7.3 Raffinement du vidage en mémoire

Nous avons vu à la section 2.3.4 que lorsqu'une variable est vidée en mémoire alors elle l'était à tout point du programme, même si cela n'est pas nécessaire. Cela est dû au fait que l'on perd de l'information lorsque l'on passe du problème d'allocation de registres au problème de coloration de graphe. Évidemment il ne faudrait charger et stocker en mémoire une telle variable uniquement aux endroits pertinent, mais cela reste un problème peu simple.

Mon maître de stage m'a déjà fourni quelques références [8] [7] [2] [5] pour approfondir ce sujet…

Bibliographie

[1] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, 1999.

[2] Peter Bergner, Peter Dahl, David Engebresten, and Mattew O'Keefe. Spill code minimization via interface region spilling. Technical report, University of Minnesota Departement of electronical Engineering, 1997.

[3] Florent Blachot. L'allocation de registres dans un compilateur industriel (stage dea). Master's thesis, Université de Grenoble, 2003.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice university, 1992.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. Technical report, Rice University, 1994.

[6] R. L. Brooks. *On coloring the nodes of a network*. PhD thesis, Cambridge, 1941.

[7] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. Technical report, Rice university and Trilogy Developement Group, 1998.

[8] Kad-Filip Faxén. Back-end isues for modern microprocessor : The state of the art. Technical report, 1997.

[9] Lal George and Andrew W. Appel. Iterated register coalescing. Technical report, Bell Labs and Princeton University, 1996.

[10] Chaitin Gregory, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register allocation via graph coloring. Technical report, Computer Languages, 1980.

[11] Allen Leung and Lal George. A new MLRISC register allocator. Technical report, New-York University and Bell Labs, 2001.

[12] Tanguy Risset. Cours de compilation, MIM, 2003-2004.

[13] STMicroelectronics. ST220 instruction set architecture. Technical report, STMicroelectronics, 2001.

[14] Alfred V., Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principes, Techniques, and Tools*. Addison-Wesley, 1986.