Étude des problèmes de spilling et coalescing liés à l'allocation de registres en tant que deux phases distinctes

Florent Bouchez

N° d'ordre : 511

N° attribué par la bibliothèque : 07ENSL0 511

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon — École Normale Supérieure de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallélisme

École doctorale de mathématiques et d'informatique fondamentale de Lyon

présentée et soutenue publiquement le 30/04/09

par Monsieur Florent BOUCHEZ

Titre :

A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases

---

Directeur de thèse : Monsieur Alain DARTE
Co-directeur de thèse : Monsieur Fabrice RASTELLO

Après avis de : Monsieur Keith D. COOPER, Membre/Rapporteur
Madame Christine EISENBEIS, Membre/Rapporteur
Monsieur Jens PALSBERG, Membre/Rapporteur

Devant la Commission d'examen formée de :
Monsieur Keith D. COOPER, Membre/Rapporteur
Monsieur Michel COSNARD, Membre
Monsieur Alain DARTE, Membre
Madame Christine EISENBEIS, Membre/Rapporteur
Monsieur Jens PALSBERG, Membre/Rapporteur
Monsieur Fabrice RASTELLO, Membre

À ma famille d'aujourd'hui,
ma future épouse,
et la famille que je construirai avec elle.

Acknowledgments / *Remerciements*

When I came to think of all the people who have helped my, accompanied me, or crossed my path when I was doing my Ph.D., it struck me that there are in fact scores of them. Now is the time to make them justice and thank them for all. Most of them are French, and after writing a whole thesis manuscript in English, I feel that I owe to my mother tongue to at least write the acknowledgments in French.

Cette thèse aura duré au final un peu plus de trois années. Si l'on y inclut le temps passé en stage de DEA, et les derniers mois avant ma soutenance, c'est au final quatre ans passés à travailler sur un même sujet, au sein d'une petite communauté de recherche qui m'a fourni un cadre de travail que je juge exceptionnel, ce qui est une chance que tout le monde n'a pas. Pour cette raison je tiens à remercier chaudement mes directeurs sans qui cette thèse n'aurait jamais existé, Alain Darte et Fabrice Rastello, qui furent à la fois des mentors qui permirent l'instauration d'un cadre de travail scientifique de grande qualité, mais aussi des personnes que l'on peut louer pour leur qualités humaines. Ils surent créer un environnement chaleureux et propice à la curiosité scientifique que je m'efforcerais de ne pas oublier si un jour c'est à moi que revient une telle responsabilité.

Merci à mes rapporteurs de thèse qui ont accepté de se plonger dans la lecture de ce manuscrit et qui m'en ont fait un retour si positif : Keith D. Cooper, Christine Eisenbeis et Jens Palsberg. En particulier, merci à Keith et Jens de s'être déplacés de si loin pour ma soutenance, de nous avoir également invités dans leurs universités respectives; ces visites furent riches en échanges. Merci à Michel Cosnard qui a accepté d'être président de mon jury de thèse malgré le peu de temps que lui laisse sa charge de la présidence de l'Inria.

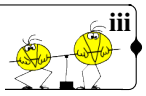
Merci à mes collègues de travail et co-bureaux pour leur présence réconfortante, compagnons d'infortune de l'aile sud de l'ENS en plein été, Benoit, Sebastian et Quentin. Avec mention spéciale pour Herr Hack dont le travail a significativement influé sur le mien et qui a supporté un an mes critiques acerbes sur sa manipulation de la langue française. Encore merci à Quentin qui a accepté la tâche ardue d'implanter, de tester, voire d'améliorer nos algorithmes et qui n'est pas encore sorti de l'auberge.

Merci aux collègues de STMicroelectronics pour toutes les journées où ils nous ont accueillis à Grenoble, pour leur efficacité et leur vision industrielle, autrement différente et complémentaire de notre vision académique, et en particulier Benoît Dupont de Dinechin et Christophe Guillon pour leur expertise indispensable et leur compétence qui ne finira jamais de m'étonner.

Merci également aux amis qui furent présents de près ou de loin durant ces années de thèse et sans qui la vie n'aurait pas été aussi joyeuse. Il y a les parisiens, David, Estelle et Nicolas, les grenoblois, Mathias et Maryline, les colocataires, Florent et bien sûr Camille rue de Marseille ; Julien toujours là pour tenter d'améliorer la beauté graphique discutable de mes figures et présentations, et Nolwenn qui partagèrent la place Colbert. Sylvain et Sophie qui m'ont supporté durant les moments sombres de la dernière ligne pas si droite que ça mais toujours là pour une partie de squash ou une virée à Saint-Étienne.

Merci à la famille Tichadou, qui m'a encouragé et accueilli comme un des leurs. Merci à ma famille, qui a toujours cru en moi et a su me laisser découvrir ma voie.

Et enfin, « last but not least », merci à ma Sophie, qui fut présente avec moi durant les deux dernières années de ma thèse, et qui m'accompagna tout au long de la rédaction de mon manuscrit, jusque ma soutenance. Sans elle, cette épreuve aurait été autrement plus difficile. Merci d'être à mes côtés.



Abtract

The goal of register allocation is to assign the variables of a program to the registers or to *spill* them to memory whenever there are no register left. Since memory is much slower than registers, it is best to minimize the spilling. However, the problem is complicated because spilling is tightly bounded with the colorability of the program. Chaitin et al. [1981] modeled register allocation as an interference graph coloring problem, which they proved NP-complete. So, there is no exact way in this model to tell whether some spilling is necessary or not, and if it is, what to spill and where. In Chaitin et al.'s algorithm, a spilled variable is removed everywhere in the program, even at places where there is enough registers, which leads to unnecessary memory transfers.

To address this problem, many authors remarked that *splitting* the live ranges of variables by inserting copy instructions creates smaller live-ranges. Hence, only part of live ranges can be spilled instead spilling "everywhere." The difficulty is then to choose the right places to split the live ranges. In practice, authors get better spill results when splitting at many program points [Briggs, 1992; Appel and George, 2001], but splitting introduces register-to-register moves to reconcile variables with sub-variables in case they are colored differently. *Coalescing* is expected remove most of these `move` instructions, but if it does not, the benefit of a better spill can be canceled out. This led Appel and George [2001] to introduce the "Coalescing Challenge."

Recently (2004), three teams discover that the interference graph of a program under Static Single Assignment (SSA) is chordal. Hence, coloring the graph becomes easy with a simplicial elimination scheme and there has been hopes that SSA would simplify register allocation. Ours were that, as the coloring was, the spilling and the coalescing might get easier to solve, as we now have a exact coloring test.

Our first goal was to better understand from where the complexity of register allocation does come, and why SSA seems to simplify the problem. We came back to the original proof of Chaitin et al. [1981], finding that the difficulty comes from the presence of (critical) edges and the possibility to perform permutations of colors or not. We studied the spill problem under SSA and several versions of the coalescing problem. The general cases were proven NP-complete but we hopefully found one polynomial result: incremental coalescing for programs under SSA. We used it to design new heuristics to better solve the coalescing problem, so that an aggressive splitting can be used beforehand.

This led us to promote a better register allocation scheme. While previous tentatives gave mitigated results, our better coalescing allowed us to cleanly separate register allocation into two independent phases: First, spilling to reduce the register pressure to the number of registers, possibly by splitting a lot; Then color the variables and perform coalescing to remove most of the added copies.

This scheme is expected to perform well in an aggressive compiler. However, the high number of splits and the increased compilation time required to perform the coalescing is prohibitive for just-in-time (JIT) compilation. So, we devised a heuristic, called "permutation motion," that is intended to be used with SSA-based splitting in place of our more aggressive coalescing in a JIT context.

Keywords: Register allocation, SSA, spilling, coalescing, complexity.

Résumé

Le but de l'allocation de registres est d'assigner les variables d'un programme aux registres ou de les « spiller » en mémoire s'il n'y a plus de registre disponible. La mémoire est bien plus lente, il est donc préférable de minimiser le spilling. Ce problème est difficile il est étroitement lié à la colorabilité du programme. Chaitin et al. [1981] ont modélisé l'allocation de registres en le coloriage du graphe d'interférence, qu'ils ont prouvé NP-complet, il n'y a donc pas dans ce modèle de test exact qui indique s'il est nécessaire ou non de faire du spill, et si oui quoi spiller et où. Dans l'algorithme de Chaitin et al., une variable spillée est supprimée dans tout le programme, ce qui est inefficace aux endroits où suffisamment de registres sont encore disponibles.

Pour palier ce problème, de nombreux auteurs ont remarqué que l'on peut couper les intervalles de vie des variables grâce à l'insertion d'instructions de copies, ce qui crée des plus petits intervalles et permet de spiller les variables sur des domaines plus réduits. La difficulté est alors de choisir les bons endroits où couper les intervalles. En pratique, on obtient de meilleurs résultats si les intervalles sont coupés en de très nombreux points [Briggs, 1992; Appel and George, 2001], on attend alors du coalescing *qu'il enlève la plupart de ces copies, mais s'il échoue, le bénéfice d'avoir un meilleur spill peut être annulé. C'est pour cette raison que Appel and George [2001] ont créé le « Coalescing Challenge ».*

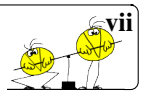
Récemment (2004), trois équipes ont découvert que le graphe d'interférence d'un programme sous la forme Static Single Assignment (SSA) sont cordaux. Colorier le graphe devient alors facile avec un schéma d'élimination simpliciel et la communauté se demande si SSA simplifie l'allocation de registres. Nos espoirs étaient que, comme l'était le coloriage, le spilling et le coalescing deviennent plus facilement résolubles puisque nous avons à présent un test de coloriage exact.

Notre premier but a alors été de mieux comprendre d'où venait la complexité de l'allocation de registres, et pourquoi le SSA semble simplifier le problème. Nous sommes revenus à la preuve originelle de Chaitin et al. [1981] pour mettre en évidence que la difficulté vient de la présence d'arcs critiques et de la possibilité d'effectuer des permutations de couleurs ou non. Nous avons étudié le problème du spill sous SSA et différentes versions du problème de coalescing : les cas généraux sont NP-complets mais nous avons trouvé un résultat polynomial pour le coalescing incrémental sous SSA. Nous nous en sommes servis pour élaborer de nouvelles heuristiques plus efficaces pour le problème du coalescing, ce qui permet l'utilisation d'un découpage agressif des intervalles de vie.

Ceci nous a conduit à recommander un meilleur schéma pour l'allocation de registres. Alors que les tentatives précédentes donnaient des résultats mitigés, notre coalescing amélioré permet de séparer proprement l'allocation de registres en deux phases indépendantes : premièrement, spiller pour réduire la pression registre, en coupant potentiellement de nombreuses fois ; deuxièmement, colorier les variables et appliquer le coalescing pour supprimer le plus de copies possible.

Ce schéma devrait être très efficace dans un compilateur de type agressif, cependant, le grand nombre de coupes et l'augmentation du temps de compilation nécessaire pour l'exécution du coalescing sont prohibitifs à l'utilisation dans un cadre de compilation just-in-time (JIT). Nous avons donc créé une nouvelle heuristique appelée « déplacement de permutation », faite pour être utilisée avec un découpage selon SSA, qui puisse remplacer notre coalescing dans ce contexte.

Mots-clés: *Allocation de registres, SSA, spilling, coalescing, complexité.*

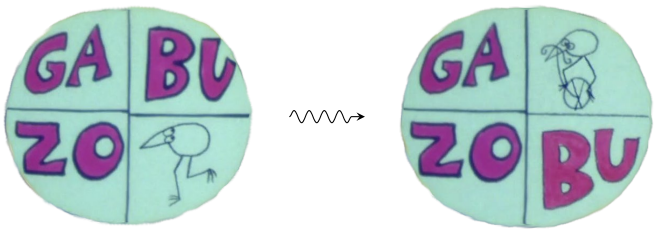


Avant-propos

J'ai fait le choix de rédiger ma thèse en anglais. Ce n'était ni par facilité, ni par vantardise, mais dans le but d'avoir un impact plus grand que si la langue de rédaction avait été le français. C'est d'ailleurs ce qui m'a permis d'avoir deux rapporteurs étrangers, ce qui est une bonne chose. Mais c'est peut-être dommage car il est sûrement important que des travaux scientifiques soient rédigés en français pour faciliter, en France, la dissémination de la science. Je me sens donc un peu coupable de ce point de vue et ai décidé que ma thèse comporterait un avant-propos en français, qui ne serait pas juste une courte traduction de l'introduction mais un petit « bonus » pour les chanceux qui connaissent la langue de Molière.

Ceux qui étaient présents lors de ma soutenance, et ceux qui en auront eu vent depuis, le savent déjà : les ordinateurs, ça marche comme les Shadoks. Ou plutôt, ça pompe comme les Shadoks puisque tout le monde sait que ces drôles de bêtes, inventées par Jacques Rouxel et dont les histoires furent narrées par Claude Piéplu dans les années soixante-dix, passent la majeure partie de leur temps à pomper, par exemple pour regonfler la lune comme l'illustrent bien les petits dessins au bas des pages de cette thèse, à côté des numéros de page. Ces Shadoks sont très intéressants car voici ce que l'on apprend au début de la série « ZO » :

> Les cerveaux des Shadoks [...] avaient une capacité tout à fait limitée. Ils ne comportaient en tout que quatre cases. Et encore c'était pas toujours vrai parce que bien souvent il y en avait de bouchées. Pour remplir les cases, déjà c'était pas facile et cela prenait un certain temps. C'est alors que commençait la difficulté parce que quand les cases était pleines, il n'y avait plus de place, et le Shadok on ne pouvait plus rien lui apprendre. Si on essayait quand même, alors obligatoirement il y avait une case qui se vidait pour faire de la place. De sorte que quand un Shadok, avec une tête pleine, voulait apprendre quelque chose, il fallait qu'il en oublie une autre. Exemple : si un Shadok avait appris à marcher avec une case, et que plus tard il ait appris trois mots avec les trois autres cases, et bien si en plus on voulait lui apprendre à faire du vélo, le Shadok ne savait plus marcher.



Et bien les ordinateurs ont un comportement très similaire à celui des Shadoks. Un ordinateur dispose également d'un nombre limité de cases que l'on appelle « registres », et qui lui servent à stocker les nombres avec lesquels il fait ses calculs. Par exemple, dans la série des processeurs x86 (dont le Pentium 4), chacun possède huit registres. À la différence des Shadoks, les ordinateurs disposent de nos jours d'une mémoire supplémentaire, beaucoup plus grande mais dont l'accès est aussi beaucoup plus lent, appelée « cache ». Si un ordinateur n'a plus de place dans ses registres mais a pourtant besoin d'une nouvelle valeur, il peut stocker temporairement une des valeurs contenue dans un registre dans la mémoire pour libérer ce dernier. Il devra

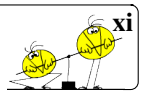
alors retourner chercher dans la mémoire la valeur évincée quand il en aura à nouveau besoin.

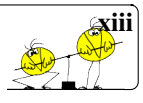
Dans le domaine de la compilation de programmes, où l'on cherche à traduire un programme écrit dans un langage dit « de haut niveau » en instructions directement compréhensibles par la machine, il nous faut allouer les variables du programme aux registres, c'est-à-dire déterminer par avance où résidera chaque variable à tout instant de l'exécution du programme. À l'instar des Shadoks, on ne peut garder en registre à un instant donné qu'au plus autant de variables que de registres disponibles. Le reste des variables doit être placé en mémoire, ce qu'on appelle le « spill ». L'inconvénient est qu'il faut du temps supplémentaire pour exécuter les nouvelles instructions de copie des variables vers ou depuis la mémoire. En général, le but de l'allocation de registres est de trouver une allocation des variables en registres et mémoire qui minimisera le temps perdu à échanger des données avec la mémoire.

Ma thèse s'inscrit dans la continuité de la recherche sur l'allocation de registres, problème largement étudié par le passé mais qui est encore un domaine actif. J'espère que ces travaux permettront aux Shadoks des générations futures[1] *d'être équipés d'un système d'allocation de cases amélioré avec transformation par SSA, spill à volonté, coalescing façon BU-GA, pompe à permutation et tout le confort actuel que pourront bientôt proposer les compilateurs modernes. Professeur Shadoko, si vous lisez ces lignes…*

[1] Hélas, leurs auteur et narrateur sont décédés les 25 avril 2004 et 24 mai 2006.

Contents



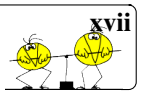


Nomenclature

(u, v)	edge between u and v, usually representing an interference, page 14
$\langle u, v \rangle$	affinity between variables u and v, page 15
χ	chromatic number of a graph, page 18
$col(x)$	color assigned to variable x, page 125
$\ldots \leftarrow a$	variable a is used by some instruction, page 12
ω	clique number, page 19
ϕ	choice function in SSA programs, page 26
$\mathcal{A}$	set of affinities, page 83
$\Omega(p)$	number of variables live at point p, page 15
Ω	Maxlive, maximum number of simultaneously alive variables, page 59
$\mathrm{def}(a)$	set of instructions which define a, page 26
$\mathrm{use}(a)$	set of instructions which use a, page 26
$w\langle u, v \rangle$	weight of the affinity $\langle u, v \rangle$, page 15
$a \leftarrow \ldots$	variable a is defined by some instruction, page 12
h	number of simultaneous holes, page 70
$s \preceq t$	s dominates t, page 27
$w(v)$	weight of variable v, representing a cost, page 59
$x[R_i]$	variable x is assigned to register R_i, page 151
R	number of registers, page 16

Acronyms

ABI	application binary interface
CFG	control-flow graph
CLI	Common Language Infrastructure
ILP	integer linear programming
IRC	Iterated Register Coalescing
JIT	just-in-time
LAO	linear assembly optimizer (compiler at STMicroelectronics)
SSA	Static Single Assignment
VLIW	Very Long Instruction Word
X3C	3-exact cover (NP-complete problem)



And the Lord came down to see the city and the tower, which the children builded. And the Lord said, Behold, the people is one, and they have all one language; and this they begin to do; and now nothing will be restrained from them, which they have imagined to do. Go to, let us go down, and there confound their language, that they may not understand one another's speech.

Bible, Genesis 11:1-9 (KJV)

1 Introduction

1.1 Program compilation

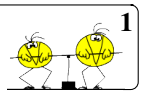
The very first computers were programmed "by hand," i.e., directly in the assembly language corresponding to their instruction set. With the growing development of new computer architectures in the 50's, machine-independent programming languages were proposed, along with the need for a program capable of converting programs written in "high-level" languages to the "low-level" language of the target machine. The first program capable of performing this task, a *compiler*, was written by Grace Hopper in 1952. Then came many others, capable of targeting multiple architectures, or accepting as input more evolved programming languages.

The most important property for a compiler is preserving the semantics of the original program. This means that, whatever the compiler used, a program should produce the same output.[1] Usually, the output of a program is the result of some computations, while other manifestations like the time required to compute the result, or the memory used, are considered as side-effects. It is often tolerated that the behavior of a program differs on the side-effects. Once it is assured that a compiler preserves the semantics of the input programs, there is still work to do on the compiler, on the "side-effects" of the compiled program. In that case, we are talking of *optimizing compilers*, i.e., compilers that also try to optimize the resulting low-level program so as to gain for instance more efficiency in speed or memory consumption, or even speed of the compiler itself.

Although new languages continue to appear and research on how to compile them is conducted, we will not study this problem in this thesis. Today, guaranteeing the semantics is usually not an issue for widely used programming languages like C. For them, there is a constant demand on optimizing compilation. The goals of an optimization are multiple and strongly depends on the context. The most preferred one is usually the speed of the compiled program, but there can also be strong needs in terms of memory use, power consumption, heat generated, code size, etc. especially in the growing context of embedded systems that have tight constraints of energy, computing power and weight.

Compilation for embedded processors can be either aggressive or just-in-time (JIT). Aggressive compilation is allowed to use a longer compile time to find better solutions. The program is usually cross-compiled, then loaded in permanent memory (ROM, flash, etc.), and shipped with the product. The compilation time is not the main issue as compilation happens only once. Furthermore, especially for embedded systems, code

[1] It should be noted that this is different from the property "a program should behave as expected by the programmer" since it generally does not...



size and energy consumption usually have a critical impact on the cost and the quality of the final product. JIT compilation is the compilation of code on the fly on the target processor. Currently the most prominent languages for JIT compilation are Common Language Infrastructure (CLI) (Microsoft) and Java (Sun). The code can be uploaded or sold separately on a flash memory, then compilation can be performed at load time or even dynamically during execution. This allows for instance to ship only one code for different platforms, or even for a platform that has different embedded architectures; then the code can be compiled for one particular processor when required, which saves a lot of space. The heuristics used for JIT compilation, constrained by time and limited memory, are far from being aggressive. In this context, trade-offs are made between resource usage for compilation and quality of the resulting code.

1.2 Register allocation

One of the most important passes in a compiler, if not the most important one, is called *register allocation*. The goal of register allocation is to map the variables of a program to physical memory locations. The compiler must indeed decide, in advance, in which locations will be held the values necessary for the computations of the program, and so for each instruction of the program. Registers are a very fast memory, hence preferred for holding these values, which are directly needed by the CPU. But there is a limited, small number of registers available in a processor, for instance only 8 registers for the IA-32 architecture (X86, 32 bits), or 64 for the ST200, a Very Long Instruction Word (VLIW) processor developed by STMicroelectronics. On the other hand, in the initial program representation, and until very late in the compiler back-end, values are stored in *variables* or *temporaries*, which are unbounded in number (see figure 1.1).

Initial *C*-like code	Assembly-like register allocated code
$a \leftarrow 18$	$R_1 \leftarrow 18$
$b \leftarrow 42$	$R_2 \leftarrow 42$
$c \leftarrow a + b$	$R_3 \leftarrow$ `add` R_1, R_2
$d \leftarrow c * b$	$R_1 \leftarrow$ `mult` R_3, R_2
$e \leftarrow -d$	$R_1 \leftarrow$ `neg` R_1

Figure 1.1: On the initial hand-written code, the programmer considers as many variables as needed. On the final machine level code, the number of physical memory resources is limited. Register allocation aims at mapping virtual variables on physical registers.

In practice, there are usually several different types of registers capable of holding different types of values: integers, floats, addresses, booleans, etc. All registers are not equivalent nor equivalently considered. For instance it can be possible to store a boolean value into an integer register but not the converse. There can be many register constraints like register aliasing (for instance, some 32-bit registers can be accessed by three aliases in X86, one for the whole register and two names emulating two 16-bit registers), or register pairing (forcing two distinct variables to be allocated to two consecutive registers).

In this thesis, we always consider only one kind of register and no such constraints. However, we will discuss in conclusion, Chapter 8, how

to solve these practical issues that cannot be left aside when compiling for actual architectures.

Since the number of variables authorized in a program is unbounded, it often happens that, on some points of the program, there are more variables than the number of registers. Some of the variables must be then be held temporarily in another memory. Usually, there is a hierarchy of memories, from the fastest and smallest to the biggest and slowest: registers, cache memory (L1, L2,...), RAM and finally hard disks. Classically, when a memory is too small to hold some information, it is virtually increased by using the next memory. This is called a "swap" if using the hard disks when there is no more space in the RAM. For the smallest memory of the hierarchy, the registers, this is a *spill*. Spilling a value in memory for future uses reduces the register pressure since stored values do not need to be kept in registers, as shown by the example Figure 1.2.

Initial C-like code	Assembly-like register allocated code
$a \leftarrow 18$	$R_1 \leftarrow 18$
	`store` $@a \leftarrow R_1$
$b \leftarrow 42$	$R_1 \leftarrow 42$
$c \leftarrow 75$	$R_2 \leftarrow 75$
$d \leftarrow b + c$	$R_1 \leftarrow$ `add` R_1, R_2
	$R_2 \leftarrow$ `load` $@a$
$e \leftarrow a + d$	$R_1 \leftarrow$ `add` R_1, R_2

Figure 1.2: The initial C-like code would need three registers to hold variables a, b and c. Spilling variable a allows to use only two registers (the @ sign symbolizes the memory address of a variable, usually a static place computed at compile time).

When a variable is "spilled" from the registers to memory, there are additional costs. The cost of the `store` and `load` operations required for the transfers to and from memory, or, if the architecture supports instructions operating with memory arguments, the increased cost of such operations, which are usually slower than those working only with registers. Hence it is usually considered that spills should be avoided as much as possible, and many register allocation algorithms try to minimize the impact of spilling.

On the impact of scheduling on register allocation. Some phases in the compiler can *schedule* the code, i.e., modify the order in which instructions are executed. This is a problem since scheduling constrains the register allocation, and conversely register allocation constrains the scheduling. An example of such a situation is depicted on Figure 1.3.

$a \leftarrow exp_1$	$a \leftarrow exp_1$	$R_1 \leftarrow exp_1$
`store` a	$b \leftarrow exp_2$	`store` R_1
$b \leftarrow exp_2$	`store` a	$R_1 \leftarrow exp_2$
`store` b	`store` b	`store` R_1
(a) Initial code	(b) Scheduling	(c) Allocation

Figure 1.3: Scheduling impacts register allocation and *vice versa*: (a) this code needs only one register; (b) if re-scheduled, the code then needs two registers; (c) if the initial code is register allocated with one register, it is not possible to re-schedule it as in (b).

Studying the impact of scheduling on register allocation is difficult, as is the problem of tuning register allocation for scheduling. People have been aware of this problem since they started to schedule code to improve software pipelining, and for instance Goodman and Hsu [1988] or Bradlee et al. [1991] proposed schemes that mixes scheduling and register allocation, or at least make the scheduling take decisions based on how would register allocation perform afterwards. Many articles on this subject have been written since then, notably the works of Ning and Gao [1993]; Eisenbeis et al. [1995] or more recently Touati and Eisenbeis [2004], Rong et al. [2005], or Kim and Lee [2006]. Kim and Moon [2007] use rotating register files, and even integer linear programming (ILP) formulation have been proposed, for instance by Nagarakatte and Govindarajan [2007]. Development in this area is mainly related to software pipelining, which we did not investigate. Hence, our work does not take scheduling into account, but focusses instead purely on register allocation. Hopefully, the results of this thesis will help research in this domain by explaining better how the register allocation works, which might gives new ideas on how to improve conditions on software pipelining so that it works well along with register allocation. To make it clear again:

In this thesis, we suppose a fixed schedule of the instructions.

1.3 Spilling & Coalescing

For a fixed schedule, the complexity of register allocation comes from two main optimizations, *spilling* and *coalescing*. Spilling decides which variables should be stored in memory to make possible register assignment, i.e., allocating all the remaining variables to registers, while minimizing the overhead of stores and loads. Register coalescing aims at minimizing the overhead of moves of variables between registers.

The difficulty of the spilling problem is in choosing which variables will be stored in memory, as well as when they will reside in memory, and where memory operations to store and fetch those variables should be placed in the program. Such operations are expensive, so it is usually advisable to minimize their number, which is a difficult problem known as the load-store optimisation problem.

Coalescing is used to reduce the number of register-to-register moves (`move` instructions). This is done either by assigning the two variables involved in a move to the same register—hence producing a instruction $[R_x \leftarrow R_x]$ that has no effect and can be removed—, or by renaming the two variables with a common name. Of course, it is not always possible to coalesce two variables, for instance, if the two variables carry different values at the same time during execution (for a dynamic point of view), or at the same place of a program (for a static point of view). Even if there is not that much move instructions in high-level programs, a lot of them are introduced during the compilation, for example when going out of a Static Single Assignment (SSA) form (a property that some intermediate program representations have, which we will introduce later, in Chapter 2), or because of register constraints on particular instructions, like the procedure `call`. Some spilling techniques involve *splitting* variables, i.e., inserting move instructions to allow different parts of variables to be assigned to different registers. This helps to spill less, but also results in the introduction of more `move` instructions in the code.

1.4 Techniques for register allocation

Early register allocation. Over the years many register allocations schemes were explored. While first approaches were local, the tendency was set towards global register allocations schemes. The former considers register allocation at basic block level, making the problem much more simpler and Horwitz et al. [1966] gives optimal algorithms for spilling and coloring for some cases (however, the general optimal local register allocation problem is NP-complete, as show by Farach-Colton and Liberatore [2000]). The latter, global register allocation, takes control-flow into account, is more complex, and Chaitin et al. [1981] proved optimal register allocation is NP-complete. But global register allocation has a larger picture to work on, which allows for better results. People got very rapidly interested in global register allocation, and suggestions for using graph coloring appeared early in the literature, for instance Yershov [1966] did and also Allen and Cocke [1976].

Introducing graph coloring in register allocation. The first to introduce a framework based on coloring of the interference graph of a program were Chaitin et al. [1981], rediscovering a coloring scheme by Kempe [1879]. In their scheme, they spill so that at most k variables are alive at the same time. Initially, $k = R$, the number of registers, and then they try to color the interference graph with R colors. If it does not work, they start again the first phase with $k = R - 1$, then $k = R - 2$, etc. until they manage to color the graph with R colors. In the same article, they also give a method to construct, for any graph, a program whose interference graph is as difficult to color, proving that this modeling of register allocation is NP-complete. Then, Chaitin [1982] refined this scheme by working directly on the interference graph also for the spilling. These two articles marked the beginning of using graph coloring based register allocators, and nearly no article on register allocation goes without citing this work since then. This elegant solution to a difficult problem is indeed appealing and led many people to work on improving it, for instance Bernstein et al. [1989] and Briggs et al. [1989], who improved the spilling and coloring. Briggs [1992] investigates the technique of live-range splitting with mitigated results. Later, George and Appel [1996] introduced their well-known Iterated Register Coalescing (IRC) scheme. Smith et al. [2004] extend the standard graph coloring technique to cope with multiple register classes and register aliasing.

Reintroducing program structure in register allocation. People also realized that, although simple and elegant it was, register allocation based solely on graph coloring lacks some insight on the structure of the program. To address this problem, Chow and Hennessy [1990] proposed a global algorithm that gives priority to frequently executed parts of the code. Other algorithms include program structure to guide graph coloring based allocators. This is for instance the choice of Callahan and Koblenz [1991] and Norris and Pollock [1994] who use the program structure and apply graph coloring to highly executed parts first. Similarly, Knobe and Zadeck [1992] use a "control tree" based on the program structure to split live-ranges between regions. Kannan and Proebsting [1995] remarked that register allocation is easy on programs that have a particular "serie-parallel" structure and propose a scheme to transform programs so that they have this property.

Even if not stated as is, the common underlying denominator of these approaches is the use of live-range splitting, as a means to focus on particular regions on the code,

as Bergner et al. [1997] do. Then, "repairing" must be performed at the boundaries, which amounts to split the variables at these points so that regions become independent. Splitting a variable means adding copies at some program points to separate its live-range into more than one connected component. This allows to spill variables only on some parts of the program and not everywhere as in the original scheme of Chaitin et al. Cooper and Simpson [1998] experiment with a "passive" live-range splitting as the aggressive splitting tentative by Briggs produced to many copies, which degraded the final result. Their strategy splits variables on demand, favoring the addition of copy instructions to the spill of a variable. Lueh et al. [2000] propose a "fusion-based" technique of incremental growth of the interference graph, starting from an inner basic block and adding the interference graphs of other regions, splitting live-ranges whenever to many variables exist.

Optimal ILP formulations. More recently, various optimal techniques using ILP have been explored. To our knowledge, the first to perform optimal register allocation where Goodwin and Wilken [1996], improved later by Fu and Wilken [2002]. This approach is also experimented by Appel and George [2001] and Barik et al. [2007]. Grund and Hack [2007] give an ILP formulation for the coalescing subproblem of register allocation.In this context also, the work of Naik and Palsberg [2004] is worth to mention, although their goal is different since they optimize the code size of the resulting program.

Linear scan allocation. With the growing proportion of embedded processor, different kinds of needs made their appearance. In particular, just-in-time (JIT) compilation aims at compiling code on the fly, and is much more constrained in time and space than aggressive, off-line compilation. Global heuristics based on local register evaluation are considered, and Poletto et al. [1997]; Traub et al. [1998]; Poletto and Sarkar [1999] introduce a new type of register allocation algorithm, the "linear scan." This one is not based on graph coloring, but instead linearizes the entire program as a unique basic block, on which local allocation is performed. This allows for a very fast algorithm, and do not need to construct a memory consuming interference graph. Improvements are made to the linear scan algorithm. Wimmer and Mössenböck [2005] introduce in it a splitting method to reduce the problem that linear scan "fills gap" of live-ranges when linearizing the program, hence pessimistically increase register pressure. Approaches still based on graph coloring are also explored for JIT compilation, for instance, Cooper and Dasgupta [2006] tailor a Chaitin-like allocator to make it run faster. Recently, Sarkar and Barik [2007] proposed an "extended" version of linear scan as a viable alternate solution to graph coloring.

Introduction of SSA in register allocation. The Static Single Assignment (SSA) form is an intermediate program representation introduced by Alpern et al. [1988] and Rosen et al. [1988]. A most important step in the introduction of SSA was made by Cytron et al. [1991] who gave an efficient method to transform a program into SSA form. SSA is appreciated in the compiler community for simplifying many compiler optimizations, for instance, Wegman and Zadeck [1991] use it to have a faster and easier constant propagation algorithm. Briggs et al. [1998] further improve the transformations into and out of SSA. A code is in SSA form when every scalar variable has only one textual definition in the program code. Most compilers use a particular SSA form, the SSA form with dominance property, which in short states that a variable must be defined before

being used. Up to now, SSA is not much related to register allocation, but we remarked that the interference graph of a program under SSA from is chordal [Bouchez et al., 2005]. Since coloring a chordal graph is polynomial, this lead to the design of new heuristics for register allocation, using the SSA form, a fact exploited by Brisk et al. [2005]; Pereira and Palsberg [2005], and Hack et al. [2006], who, independently, made the same observation about the SSA interference graphs being chordal. Following the idea of using SSA for register allocation, Pereira and Palsberg [2008] introduce their "puzzle-solving" technique, and Hack [2007] wrote his Ph.D. thesis.

A few words towards simplicity in register allocation. We presented here some of what we believe to be the most important steps in register allocation, from its earliest developments up to now. Many register allocation schemes were invented and described in the literature during this time. However, it is quite hard, taking any two schemes, to know precisely how well one performs compared to the other. Usually, authors compare their algorithm to what they think as "classical" register allocation algorithm that are known to work "quite well." However, this is not always the case, and among the many existing allocators or improvements of allocators, not so many are implemented and used in practice. Cooper et al. [2005] remarked for instance that the allocator proposed by Callahan and Koblenz [1991] was implemented only once and no assessment were reported in the literature, so they did a thorough work of implementation and comparison with the Chaitin et al. algorithm with improvements by Briggs. More recently, Cooper et al. [2008] did a similar work comparing the priority-based algorithm of Chow and Hennessy [1990] with Chaitin-Briggs. We view this situation as a clear example that the simpler and more elegant ideas are the ones that make their way through all the others. Very smart but very complicated schemes are appreciated by the community, but make life harder for others whenever they want to compare their algorithms. Hence it is often seen that improvements on a particular scheme are compared to the original scheme, but not against each other. In practice, someone who wants to implement a compiler will then have trouble deciding whichever scheme is the best, and will obviously choose the ones that are simpler, both from a conceptual and an implementation point of view. It is our belief that any new scheme, idea, or improvement of an existing scheme should be simple, or at least easy to understand. We think that this is one of the reasons that the SSA form is getting more and more appreciated today, since it simplifies many compiler optimizations. This is also probably the same reason why "linear scan" allocators are very popular nowadays. Some graph coloring improvements are today so complicated that, by contrast, the simplicity of the linear scan algorithms makes people have more faith in them. And it is not uncommon to hear people say sentences like the following:

> "Essentially, although graph coloring in register allocation was very popular in the 90's [...] the existing graph coloring algorithms neither produce faster code, nor have faster compilation time than the [linear scan] algorithms already in use.[2]"

This is not a point of view that we share, as, conceptually, a linear scan allocator has less access to global information than a graph coloring based allocator. Of course, this is not true with original graph coloring schemes, but splitting techniques can make interference graphs much more precise. However, our point here is that we do think

[2]Excerpt from a review of our article on coalescing [Bouchez et al., 2008] when in was rejected at CC'08.

that simpler schemes are more popular, and are easier to modify and improve. Following this idea, we already pointed out that the SSA form simplifies the shape of the interference graph by making it chordal. We wanted to investigate this area, remarking that, by using SSA (for instance, but not only), the problem of register allocation can be cleanly separated into two phases, hopefully making it simpler to deal with.

1.5 About this thesis

In this thesis, we restrict our interests in register allocation to graph coloring schemes in the line of the original algorithm by Chaitin et al. [1981]. Since the NP-completeness proof of Chaitin et al., people take for granted that register allocation is a difficult problem. Most graph coloring schemes intermix all subproblems of register allocation in a common phase: assign and allocate variables to registers while minimizing the spilling overhead and coalescing unnecessary move instructions. Our discovery that SSA interference graphs are chordal shows that, in fact, the complexity does not come from the "coloring," which is a misinterpretation of Chaitin et al.'s proof. In fact, it shows that register *assignment* is easy: if there are enough registers, splitting live-ranges as does SSA is sufficient and a greedy algorithm manages to color the interference graph. This re-motivated the design of register allocation based on graph coloring as a scheme in two parts: First, reduce the number of alive variables by spilling so that they fit in the register available, this is *register allocation*. Second, map variables to individual registers, potentially by splitting variables, this is *register assignment*.

This idea is not new, and was already explored by a few people in old articles, for instance by Cytron and Ferrante [1987] and Knobe and Zadeck [1992], then more recently by Appel and George [2001] or Hack [2007]. The original algorithm of Chaitin [1982] is really simple and works well, so it is not surprising that the two phases were performed in only one. But a lot of improvements to this original scheme did not make their way to compilers. The algorithms get too intricate and complicated because the two phases are not cleanly separated, and the same is true for other allocation schemes as well. Still, traditionally, spilling and coalescing are done in a common phase. Why is graph-based register allocation still nearly always performed in only one phase? We think there are three main reasons for this:

- Spilling is strongly dependent on the coloring property of the interference graph. And the coloring is a difficult problem, hence heuristics that give an actual coloring are used: spills are done until the coloration succeeds, i.e., *one knows the allocation is correct whenever one has a working assignment of variables.*

- Coalescing changes the structure of the interference graph. Aggressive coalescing might induce more spilling, hence cannot be in a separate later phase. Conservative coalescing guarantees that no additional spilling will be necessary, but it can help with the coloring, reducing the number of colors needed. This was remarked by George and Appel [1996] and used in a register allocation framework by Vegdahl [1999]. Hence, *coalescing can reduce the number of spills required.*

- Conservative coalescing is usually difficult to perform effectively if there is a high number of `move` instruction. So, algorithms were designed to make trade-offs between spilling and splitting. This is easier to balance if there is only one phase in which *splitting can be done on demand.*

These reasons are now obsolete because we now know that the interference graph on an SSA program is chordal. A chordal graph needs as many colors as the size of its biggest clique, i.e., its biggest complete sub-graph. For an SSA program, this corresponds to the maximum number of variables simultaneously alive. Moreover, no coalescing can reduce this number since no two nodes of a complete graph can be coalesced: they must reside in different registers. We now have an exact test of the number of registers required for the allocation, provided that the live-range structure is fixed as the one generated by the SSA form. In general, given R registers and supposing any splitting technique that gives, as does SSA, the chordal property to the interference graph, register allocation can be decomposed into two different phases:

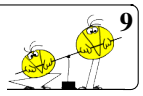
1. Spill variables so that there are at most R simultaneously alive variables at each point of the program.

2. Split, then color the interference graph while performing conservative coalescing to preserve the R-colorability.

There remains the last reason why phases were not separated: known coalescing techniques do not cope well with too many move instructions. This problem was already known to Briggs [1992] when he tried aggressive live-range splitting. Because of this, Cooper and Simpson [1998] prefer to perform splitting on demand in order not to create too many copies. More recently, this problem bothered Appel and George [2000] so much that they launched the "Optimal Coalescing Challenge." Hence, having good coalescing strategies was the last missing piece of the puzzle of register allocation in two phases. For this reason, we spent a lot of time working on different variants of this problem in this thesis, both on the complexity and heuristic points of view. Finally, we found satisfying strategies that allow us to safely state that the first phase of register allocation needs not to worry anymore about introducing too many copies. The insight given by three years worth of research and this thesis statement is that:

A two-phase register allocator is simple and efficient.

Outline of this thesis. In Chapter 2, we introduce the necessary definitions of the concepts used in this thesis. We also give the proofs of two important results of this thesis: that the interference graph of a program under SSA form is chordal, and that chordal graphs are greedy-colorable, i.e., colorable using a simple greedy scheme. This lead us to ask the question why SSA programs were not covered by the NP-completeness reduction of Chaitin et al. [1981], which reduces register allocation to Graph k-Coloring. In Chapter 3, we come back to this proof and extend it to cover more cases, in particular involving live-range splitting. In this chapter, we outline the importance of critical edges for the complexity register allocation. We study in Chapter 4 whether SSA also simplifies the spill "everywhere" problem, a simplification of the more general spill problem often used in register allocation schemes, and find most of the studied problems NP-complete. In this chapter, we differentiate two situations depending on whether spilled variables need to reside temporarily in registers when stored and loaded from memory or not. We continue our study of the complexity of register allocation in Chapter 5, which is devoted to the coalescing problem. This is the first thorough complexity study of the different coalescing strategies used in the literature. Using this work, we improve existing coalescing techniques in Chapter 6, finding surprising results in which our advanced conservative strategy outperforms all strategies based on aggressive schemes. Chapter 7 introduces a strategy different from coalescing to



remove the copies inconveniently placed on control-flow edges when going out of SSA *after* register allocation has been performed. We introduce there our technique of "parallel copy motion," a fast and efficient method designed for JIT compilation. Finally, we conclude in Chapter 8 after discussing practical considerations of actual architectures that need to be taken into account in a register allocation algorithm.

GROUND, n. Like mattresses, only harder.

2 Grounds

In this chapter, we define the notations, vocabulary and basis for the next chapters. First, we define generalities about programs and their interference graphs. Then we discuss the coloring of the interference graphs, along with some interesting structures of interference graphs. We also see what is usually done in practice whenever there are too few colors to perform register allocation. Finally, we introduce the Static Single Assignment (SSA) form and its effects on interference graphs for register allocation; this leads us to present two of our results: that interference graphs under SSA are chordal and that chordal graphs are what we call "greedy-colorable."

2.1 Basis for register allocation

Register allocation deals with programs, variables and registers. We designate by "program" what is in fact usually called a "function" or "procedure" in the programmers' minds. Indeed, we will not include inter-procedural analysis issues in our studies. The variables are virtual value holders used in programs to perform computations, while the registers are their equivalent physical counterparts. The goal of register allocation is to allocate the virtual locations to either the physical ones or to the main memory, so that the processor can actually perform the desired computations.

2.1.1 Programs and control-flow graphs

Definition 2.1. An *instruction* is an atomic operation which possibly uses some variables and possibly defines other variables.

Example.

Instruction	defines	uses	effect
$a \leftarrow 0$	$\{a\}$	$\emptyset$	put the value 0 in variable a
$a \leftarrow b + c$	$\{a\}$	$\{b, c\}$	put the sum of b and c in a
`print` b	$\emptyset$	$\{b\}$	display the value inside b
$a, b \leftarrow$ `load64` c	$\{a, b\}$	$\{c\}$	load the 64-bits value at memory address c into 32-bits variables a and b
`test` $a \neq 0$	$\emptyset$	$\{a\}$	test if the value in a is null

Note that the `test` instruction defines in fact a boolean value, but which is not in the same register class as a hence it is not considered here.

By convention, in the examples given in this thesis, the notation "$\leftarrow$" means that: if a variable is on the left-hand side, it is defined by the instruction; if a variable is on the right-hand side, it is used by the instruction. The notation "..." inside instructions

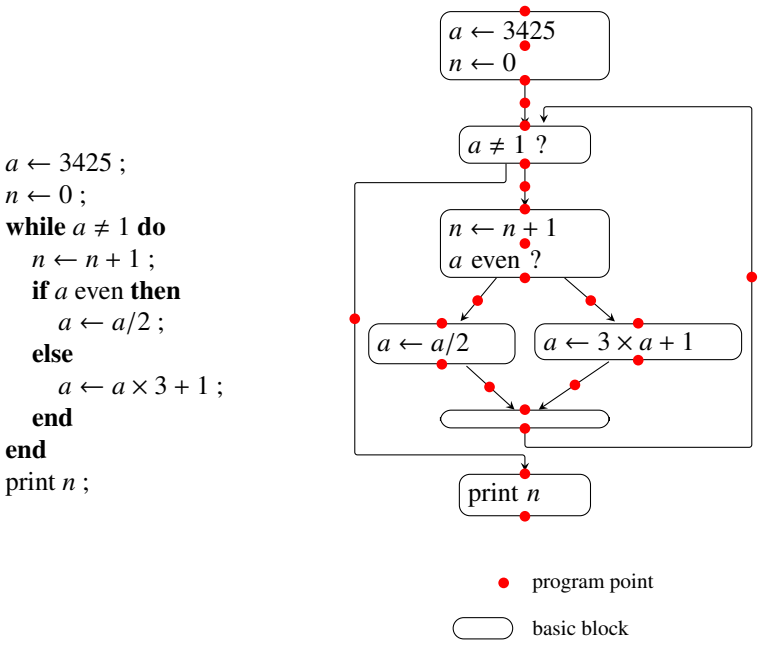


Figure 2.1: Example of a program and the corresponding control-flow graph.

means "something"; as an example, $[a \leftarrow \ldots]$ means variable a is defined using some value or calculation, and $[\ldots \leftarrow a]$ means a is used in some instruction (which does *not* necessarily define some another variable).

Definition 2.2. A *program* is a set of instructions linked by flow edges. An edge from s to d means that instruction d can be executed after s. s is called the *source* and d the *destination* of the edge.

Definition 2.3. A *basic block* is a maximal sequence of instructions without branch: there is no other leaving or entering path possible in the middle of a basic block. A program can be represented by a *control-flow graph (CFG)*, which is a graph where the vertices are the basic blocks and the oriented edges the possible paths during the execution of the program.

We will now define "program points," i.e., points of the program where, hypothetically, the program could be stopped and the state of the machine could be inspected. Hence instructions are not considered as "program points" since the state of the machine is not well-defined—is the new variable already defined? are the arguments already used?—, but points between two instructions are program points, entries and exits of basic blocks also, and even points on control-flow edges.

Definition 2.4. A *program point* is any point of the CFG which is *not* an instruction, i.e., any point on a possible execution path before or after an instruction.

The first program point of a program—the one before the first instruction—is called the *entry* or *root* of the program. A program point with more than one successor instruction is called a *branch*; a program point with more that one predecessor instruction is called a *join*.

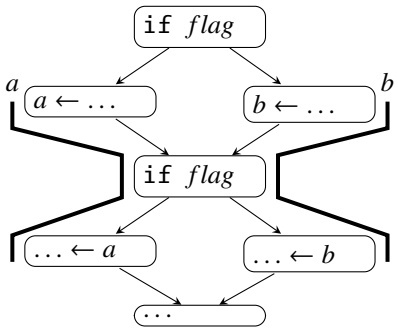
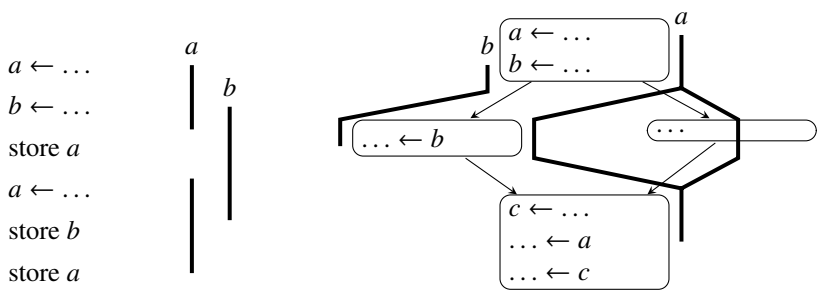


Figure 2.2: Non-strict program.



(a) Linear code

(b) More general CFG

Figure 2.3: Live-ranges of variables—thick lines—on two different examples of code.

Figure 2.1 shows an example of a program with the corresponding CFG, basic blocks, and program points. The point after $[a \neq 1]$ is a branch and the one at the beginning of the empty basic block (containing only a jump to the conditional branch) is a join.

It is often assumed that, for each use of a variable, the variable has been defined before the use. While this should be *dynamically* the case, i.e., during the execution of the program, this property is hard to check *statically*, i.e., during compilation.

Definition 2.5. A program is *strict* if for each variable and each use of this variable, there is a definition of this variable on any static control path—a path following the control-flow edges—from the start of the program to this use.

See Figure 2.2 for an example of a correct non strict program: dynamically, the execution flow only chooses the left paths or the right paths but cannot mix both. But there exist *static* paths taking the left path then the right one or the converse.

Unless stated otherwise, we will always assume strict programs.

2.1.2 Live-ranges, interference graph

The goal of register allocation is to allocate variables to memory locations, in particular registers. These are the fastest available on a processor and hence preferred over main memory. However, they are in limited, small number, and each register can hold only one value at a time. Some variables may be placed in the same register under certain

conditions, for instance if they are not live at the same time. In practice, the converse "interference" property is used:

Definition 2.6. Two variables *interfere* if they cannot be stored in the same register.

From the definition of interference, we can deduce that two variables interfere if and only if (iff) they "exist" at the same time and carry different values. However, these notions are dynamic in essence while compilation is static. In practice, relaxed definitions of the interference are being used instead of this "ultimate" one. We will define for the first condition the notion of "live-range," the domain where a variable exists statically. As for the second condition, it is in general difficult to know whether two variables carry the same value or not, so this condition is usually left aside, except for very simple cases.

The life time of a variable is the set of points where this variable has been defined previously and will be used in the future. Whenever a variable is not alive, it is dead. Figure 2.3 represents live-ranges on two examples of code: a linear code—for instance inside a basic block—and a more general code.

Definition 2.7. On a strict program, a variable is *alive* at a program point p iff there is a static path from p to a use of a which does not go through a definition of a.

The *live-range* of a variable a, live(a), is the set of program points where a is alive. These are the points between the instructions defining a, def(a) and the instructions using a, use(a). It is a sub-graph of the CFG. A variable is *live* on any program point of its live-range, and *dead* otherwise.

Using the live-ranges, it is possible to calculate easily a relaxed notion of interference.

Definition 2.8 (Relaxed interference)**.** Two variables *interfere* iff their live-ranges intersect.

This definition finds more interferences than the "ultimate" interference definition, as shown by the example Figure 2.2: on this non-strict program, the two live-ranges of a and b intersect but they can nevertheless share the same register since they are never *dynamically* alive at the same time; a and b do *not* interfere.

The relations of interference can be represented using a graph:

Definition 2.9. The *interference graph* $G = (V, E)$ of a program is an undirected graph where each vertex $v \in V$ corresponds to a variable of the program. There is an *interference* $(u, v) \in E$ iff u and v interfere.

Chaitin et al. [1981] proved the following lemma so that the notion of interference for strict programs gets very easily computable: one just needs to check it at definition points.

Lemma 2.10. *For a strict program, the live-ranges of two variables intersect iff the live-range of one contains a definition of the other.*

Chaitin et al. [1981] proposed to refine their use of the "relaxed" interference when building the interference graph by saying that if u is defined by the copy $[u \leftarrow v]$, then no edge is added between u and v in the graph, since they obviously have the same value.[1] Hence it is possible to have a different definition of interference:

[1] Note however that u and v might still interfere, for example if u is defined multiple times. In that case, an edge between u and v will be added anyway sooner or later.

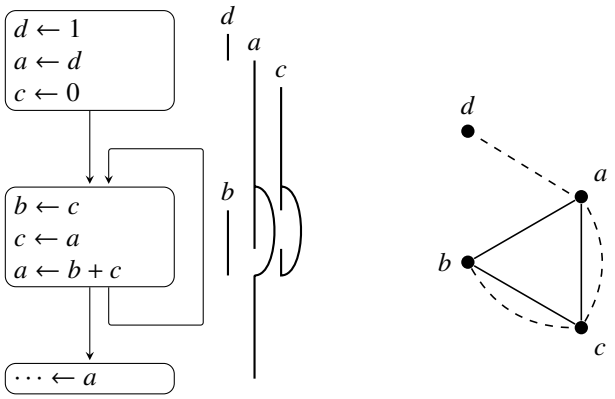


Figure 2.4: Program with the live-ranges and the corresponding interference graph. Interferences are represented with plain edges and affinities with dashed ones.

Definition 2.11 (Chaitin's interference)**.** For a strict program, Two variables u and v *interfere* iff the live-range of u contains a definition of v different than $[v \leftarrow u]$, or the live-range of v contains a definition of u different than $[u \leftarrow v]$.

The interference graph will depend on the definition of interference chosen; the more refined it is, the fewer "false" interferences there will be in the graph.

> Note: In this thesis, some theorems or properties rely on the structure of the interference graph. Hence, the notion of interference chosen can be important for the correctness of some algorithms, and a definition of interference cannot always be traded for another without checking that does not invalidates proofs. For instance, with Definition 2.8, a variables alive at one program point form a clique, while this is not true with Definition 2.11.

In addition to interferences, usually represented with solid lines, each copy instruction $[u \leftarrow v]$ is represented by an affinity, usually shown using a dashed line in the interference graph. If both variables are assigned to the same register, the corresponding assembly instruction [`move` u, v] can be removed from the program.

Definition 2.12. An *affinity* $\langle u, v \rangle$ between variables u and v in the interference graph expresses the preference for these variables to share the same color (register).

Affinities can also be weighted to represent a dynamic execution count of the copy instructions. In that case, the weight of an affinity between u and v is usually denoted $w\langle u, v \rangle$.

Figure 2.4 gives an example of interference graphs of programs, with affinities between variables linked by a copy instruction. Affinities between adjacent vertices are represented but cannot be coalesced: they are called *constrained* affinities.

2.1.3 Maxlive

Definition 2.13. Given a point p of the CFG, *Live* is the number of variables simultaneously alive at p, represented by symbol $\Omega(p)$. *Maxlive* , denoted by Ω, is the maximum of $\Omega(p)$ over all points p of the CFG.

Figure 2.5 illustrates the definition of Maxlive on the straight line code of Figure 2.3. Maxlive will be an important indicator to decide whether it is possible to

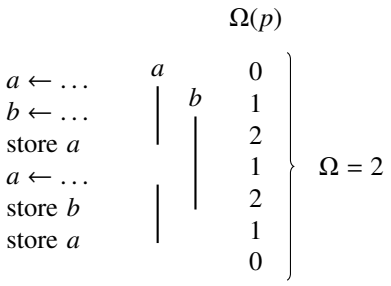


Figure 2.5: Number of variables in Live at each point and Maxlive, the maximum over all of them.

allocate all variables to registers or not. Here, we defined Maxlive with the relaxed definition of interference in mind (Definition 2.8): two variables interfere if they are alive at the same time. With this definition, and for a strict program, Maxlive is a lower bound on the number of registers required to store all variables of the program.[2] Indeed, there is at least one program point p where $\Omega(p) = \Omega$; On this point, every variable is alive: they all interfere, meaning that one needs Ω registers for this point.

If considering Chaitin's interference, in which copies of the same variable do not count (Definition 2.11), Maxlive is not a lower bound on the number of registers required anymore. If one still wants this property, $\Omega(p)$ should be defined as "the number of registers required to allocate all variables of p," which is more complicated than just counting the variables alive at p.

2.2 Coloring the interference graph

In the graph coloring problem, the goal is to assign different colors to adjacent vertices. Given a valid coloring of an interference graph, it is possible to view the colors as registers, meaning that two interfering variables are in different registers. This gives a valid register allocation for the program provided that less that R colors are used, where R is the number of registers available.

Definition 2.14. A *coloring* of the interference graph is a function *col* on the nodes such that $col(a) \neq col(b)$ whenever a and b interfere. *col* is a k-coloring if it uses at most k different values. An R-coloring of the interference graph gives a valid register allocation for a program.

Notice that, in the interference graph model, each variable is traditionally considered as an atomic object, i.e., it has a single color, meaning that it will be placed in the same register on all its live-range. In this context, the first problem considered is logically the following:

How to know if there are enough registers to allocate all variables?

We will now present the traditional way to answer this question, then what can be tried if the answer to this question is negative.

[2] This is false for a non-strict program: see Figure 2.2 again, the same register can hold both a and b since they are never *dynamically* alive at the same time.

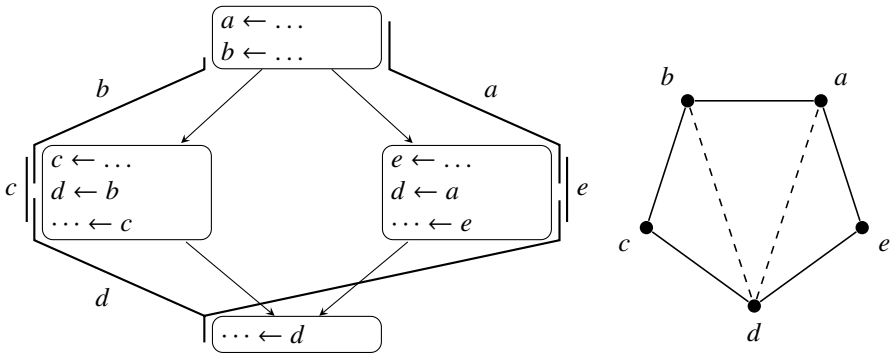


Figure 2.6: A code with $\Omega = 2$ but nevertheless not 2-colorable because the interference graph is a cycle of odd length.

2.2.1 Testing if R registers are sufficient

2.2.1.1 Conditions on Maxlive

Since there is at least one point in the program where Live is equal to Maxlive, the condition $R < \Omega$ is sufficient to know it is *impossible* to allocate the program without modifying it, i.e., *spilling* some variables to memory is necessary as we will explain later. What about the condition $\Omega \leq R$? Unfortunately, this condition is not sufficient in the general case as shown by Figure 2.6. The program of Figure 2.4 needed also three registers even if there were only two variables alive at the same time. Chaitin et al. [1981] proved in fact that the interference graph of a program can be any graph, hence the problem of allocating a *unique* register to each variable of a program reduces to Graph k-Colorability, which is NP-complete. The proof is analyzed in details in the next chapter, Section 3.1.1, and its validity is discussed whenever more freedom is allowed, for instance whenever variables can reside in different registers during their lifetime.

2.2.1.2 A coloring heuristic: Chaitin et al.'s simplification scheme

Since graph coloring is NP-complete, Chaitin et al. [1981] used a simple scheme invented by Kempe [1879] to color the interference graph with k colors.The algorithm rely on the following "simplify" rule to assign colors to variables: *A node x with fewer than k neighbors is always colorable no matter how $G \setminus \{x\}$ is colored.* It can thus be removed (simplified) from the graph and pushed on a stack. If this *simplify phase* removes all nodes, the graph is k-colorable. Indeed, in a second "select" phase, each node can be popped from the stack and colored with one of the colors not used by its neighbors previously popped, which are fewer that k. An example of execution of this algorithm for $k = 3$ is given on Figure 2.7: initially, only the node f with degree 2 can be simplified because all other nodes have degree 3, but eventually, all of them can be simplified after the simplification of some of their neighbors. However if at one point of the simplify phase, the degree of every node in the remaining graph is at least k, the coloring fails. It does not mean that the graph is not k-colorable, only that we did not find a k-coloration. This can be the case even for simple graphs such as a cycle of even length, see Figure 2.8.

During this thesis, we remarked that this greedy heuristic defines without ambigu-

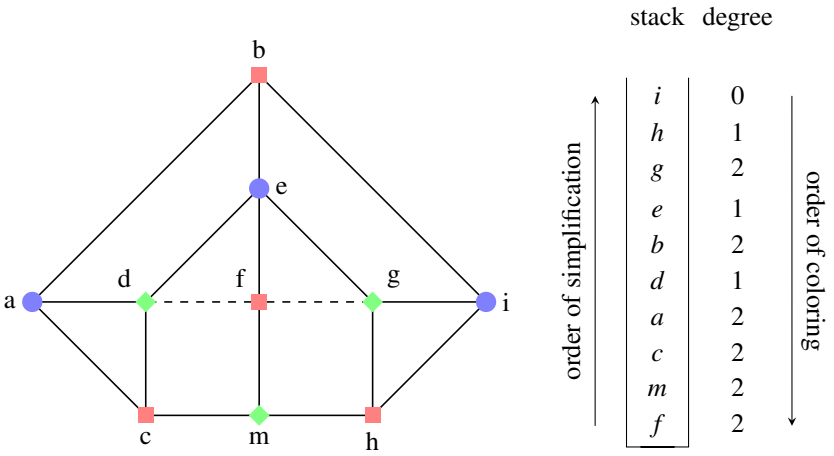


Figure 2.7: Example of Chaitin et al.'s simplification scheme with 3 colors.

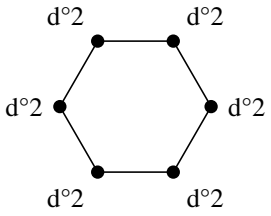


Figure 2.8: A 2-colorable graph, but the simplification scheme of Chaitin et al. fails since every node has degree equal to 2.

ity a class of graphs, to which we gave the name of greedy-k-colorable graphs, i.e., graphs colorable with k colors with this heuristic. We will define cleanly these graphs in Section 2.2.2.5, along with Function `Is_kGreedy`, a pseudo-code for the greedy heuristic. Greedy-k-colorable graphs are not the only interesting class of graphs for register allocation, and we will now introduce the graph structures which we found the most interesting for interference graphs.

2.2.2 Interesting graph structures

In this section, we recall some particular graph structures that appear as interference graphs under certain conditions. All these structures have some interesting properties in our context of deciding whether R registers are sufficient or not.

2.2.2.1 k-colorable graphs

This is the most general class of graphs. A graph G is *k-colorable* if it is possible to color it with at most k colors. In general, the minimum number of colors required to color G is the *chromatic number*, denoted by $\chi(G)$. Hence, a graph G is k-colorable for any k greater or equal to $\chi(G)$. For this class of graphs, it is NP-complete to decide, for a given integer K, if $\chi(G) \leq K$ [Garey and Johnson, 1979, Problem GT4].

2.2.2.2 Cliques

Definition 2.15. A *clique* is a complete graph, i.e., for each two nodes u and v there is an edge (u, v).

Cliques are the most restrictive graphs in terms of coloring. Clearly, a clique of size k needs exactly k colors. Hence, knowing that a graph G contains a k-clique is an interesting fact since it shows that $\chi(G) \geq k$. A useful trait for the colorability of a graph is its *clique number*, $\omega(G)$, the size of its largest clique. In the literature, *perfect graphs* are defined as graphs for which the coloring number equals the clique number, i.e., $\chi(G) = \omega(G)$ Golumbic [1980].

2.2.2.3 Interval graphs

Definition 2.16. An *interval graph* is the intersection graph of a family of intervals.

Theorem 2.17. *The interference graph of a basic block with one definition per variable is an interval graph.*

Proof. In a basic block, each live-range of a variables is a connected component if there is only one definition for that variable. Moreover, these live-ranges are sub-intervals of the basic block, starting at the definition—or at the beginning of the basic block if they are live-in—and ending at the last use—or the end of the basic block if they are live-out. □

Interval graphs are perfect graphs: as explained before, their coloring number equals the size of their largest clique. For basic blocks, this means it is possible to compute the number of registers required by performing a "scan" from the top to the bottom of the basic block while keeping a set of the live variables: the clique number is the maximum size of the set.

2.2.2.4 Chordal graphs

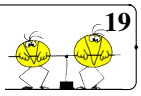
Definition 2.18. An undirected graph is *chordal* if every cycle of size at least four has a chord (edge between two non adjacent vertices of the cycle).

Chordal graphs are sometimes called "triangulated graphs" because the chords in cycles make a lot of small triangles, as shows the example on Figure 2.9. Like interval graphs, chordal graphs are perfect. Another characterization of chordal graphs uses "simplicial" vertex and "perfect elimination schemes."

Definition 2.19. A *simplicial* vertex is a vertex whose neighbors form a clique. A *perfect elimination scheme* is an ordering $\sigma = \{v_1, v_2, \ldots, v_n\}$ of the nodes such that each v_i is a simplicial vertex of the induced subgraph $G_{|\{v_i,\ldots,v_n\}}$.

A graph is chordal iff it has a perfect elimination scheme, moreover, any simplicial vertex can start a perfect scheme [Golumbic, 1980, Thm. 4.1]. This means that, if G is chordal, one can remove successively simplicial vertices until the graph is empty. It is then easy to color the nodes in the reverse order of their simplification [Fulkerson and Gross, 1965], or, as we will see in Section 2.2.2.6, more simply with the greedy simplification scheme of Chaitin et al.

Another equivalent definition [see Golumbic, 1980, Thm. 4.8] uses the tree representation:



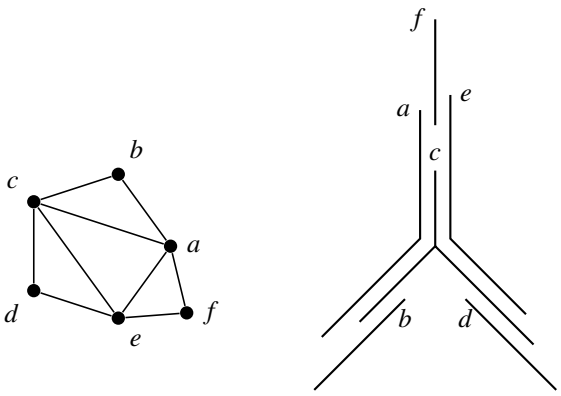


Figure 2.9: Example of chordal graph with its representation as subtrees of a tree.

Definition 2.20. A *chordal graph* is the intersection graph of a family of subtrees of a tree.

An example of chordal graph with its subtree representation is given on Figure 2.9. Using the tree representation, is is easy to color the graph also using a "scan" as for interval graphs, but in our case the scanning starts at the root and stops at the leaves of the tree. To make it short, we say that a k-colorable chordal graph is *k-chordal* .

2.2.2.5 Greedy-k-colorable graphs

Another fundamental class of graphs for Chaitin-like register allocation is what we call *greedy-k-colorable* graphs. These are the graphs k-colorable using the greedy simplification scheme of Chaitin et al. [1981] introduced in Section 2.2.1.2. For instance, the graph given as example on Figure 2.7 was a greedy-3-colorable graph.

Definition 2.21. A graph G is *greedy-k-colorable* iff there is no subgraph G' of G such that each node of G' has degree at least k in G', i.e.:

$$\nexists\, G' \subseteq G \mid \forall x \in G',\ d_{|G'}(x) \geq k$$

The following theorem links the name of this class of graphs to the greedy coloration scheme of Chaitin et al. given by Function `Is_kGreedy` (page 21).

Theorem 2.22. *A graph G is* greedy-k-colorable *iff Function* `Is_kGreedy`*(G) succeeds.*

We recall again the idea of the algorithm. While this is possible, remove a vertex of degree strictly less than k in the current graph. Indeed, whatever the coloring of the current graph, there will always be at least one color available for this vertex. Hence we need to prove that a graph is greedy-k-colorable iff this elimination scheme removes all vertices. This definition seems non-deterministic but, for a greedy-k-colorable graph, the order in which vertices are removed is not important: removing a vertex with degree $< k$ is never a bad decision for coloring. Here is a formal proof of the theorem.

Proof. $\Leftarrow$ by contraposition: suppose there exists G' subgraph of G such that $\forall x \in G',\ d_{|G'}(x) \geq k$, then, none of the nodes of G' will ever be simplified by the greedy

Function `Is_kGreedy(G)`

Data: Undirected graph $G = (V, E)$; $\forall v \in V$, degree[v] = #neighbors of v in G, k number of colors

```
1 stack = ∅ ; worklist = {v ∈ V | degree[v] < k} ;
2 while worklist ≠ ∅ do
3     let v ∈ worklist ;
4     foreach w neighbor of v do
5         degree[w] ← degree[w]-1 ;
6         if degree[w] = k − 1 then worklist ← worklist ∪ {w}
7     push v on stack ; worklist ← worklist \ {v} ;        /* Remove v from G */
8 if V = ∅ then return TRUE else return FALSE
```

simplification scheme. Indeed, the degree of a node can only decrease during simplification, hence at best all nodes not in G' can be simplified.

$\Rightarrow$ by induction on the number of nodes of the greedy-k-colorable graph G. If G has only k nodes, they can all be simplified since each node has at most $k-1$ neighbors. Suppose that a greedy-k-colorable graph with $n-1$ nodes can be simplified. Let G be a greedy-k-colorable graph with $n > k$ nodes. Then there is at least one node $x \in G$ such that $d(x) < k$ by definition. By simplifying (removing) this node, one gets G' that is greedy-k-colorable and with $n-1$ nodes, hence G can be simplified.

Finally, a greedy-k-colorable graph is k-colorable because it is possible to color its vertices in the opposite order of their removal, assigning to each vertex a color not used by its already-colored neighbors: this is possible because there are at most $(k-1)$ such neighbors. This scheme is exactly the coloring heuristic used in Chaitin-like approaches. □

2.2.2.6 Orderings of graphs structures

$$k\text{-colorable interval graphs} \subsetneq k\text{-chordal} \subsetneq \text{greedy-}k\text{-colorable} \subsetneq k\text{-colorable}$$

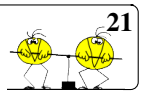
The last inclusion is trivial, the first also since an interval is a particular subtree with no branch. Example for the inequalities can be found in figures previously seen: the graph on Figure 2.8 is 2-colorable but not greedy-2-colorable; the same graph is greedy-3-colorable but not chordal (for any k) since it is a chordless cycle of size 6; Figure 2.9 shows a chordal graph which cannot be represented as an interval graph. Finally, the middle inclusion is proved by the following property of k-chordal graphs.

Property 2.23. *If G is a k-colorable chordal graph, it is greedy-k-colorable.*

Proof. Any chordal graph G has at least one simplicial vertex[3] [Golumbic, 1980], i.e., a vertex v whose neighbors form a clique: v and its neighbors also form a clique, and if G is k-chordal, it has no clique of size $k+1$. Thus, v has at most $k-1$ neighbors and can be removed (simplified) from the graph. The remaining graph is still k-chordal and the same argument applies. Thus, G is greedy-k-colorable. □

This property is one of the early contributions of this thesis, and is of much interest since it implies that Chaitin-like register allocators provide an solution whenever the interference graph of the program is R-chordal. We will see in Section 2.3 a case where this property is particularly interesting.

[3] Actually, it has at least two simplicial vertices.



2.2.3 What to do if R registers are not sufficient?

Chaitin et al.'s greedy heuristic can tell that R registers are sufficient to color the interference graph. If the heuristic fails, the goal is to modify the program so that the interference graph becomes greedy-R-colorable. In most of the cases, some nodes need to be removed from the graph. This is necessary if $\Omega > R$, for instance, since that means there is a Ω-clique in the graph, i.e., a complete sub-graph of Ω nodes.[4]

Variable spilling: In order to remove nodes from the graph, some variables are transferred—*spilled*—to memory. That way, they do not need any register to hold them at times where they are in memory. There are two problem with spilling: first, operations working with operands in memory are slower that those working with operands in registers; second, instructions to transfer values to and from memory (`store` and `load`) need to be inserted in the program, which degrades performance and uses new variables which need to be allocated: this creates new nodes in the interference graph, hopefully simpler to color that the ones spilled since their live-ranges are very short.

Variable splitting: Another technique to make the graph colorable, less powerful that spilling but also cheaper, is variable *splitting*. Let us give the intuition for a variable a on a basic block. a can be split into two variables a and a' by inserting a copy instruction $[a' \leftarrow a]$ somewhere in its live-range, on a program point. Then, subsequent uses of a are replaced by a' in the code: on the basic block, all uses before $[a' \leftarrow a]$ still reference a, but the ones below reference a'. Hence, a and a' are not alive at the same time and they might be placed in different registers.

Splitting live-ranges is more complicated on a general CFG since the consistency must be kept at join points: suppose variable a is split in the "else" part of a conditional, there is an ambiguity after the conditional: which of a or a' should be used? None of them. Either way would break the semantic of the original code: if coming from the "then" part, the a should be used, and if coming from the "else" part, it is a' which should be used. A possibility would be to restore back a' in a before leaving the "else" part, by inserting $[a \leftarrow a']$. More generally, to split a variable a into a' on a subset of its live-range, one has to insert $[a' \leftarrow a]$ at each program point where a path enters the subset, and $[a \leftarrow a']$ at each program point where a path leaves the subset.

Splitting variables allows them to be stored in different registers at different points of their lives, which simplifies the coloring: smaller live-ranges may have fewer interferences, hence are easier to simplify using Chaitin et al.'s scheme. For instance, we used two examples of code where Maxlive equals two, but three colors are required anyways in Section 2.2.1.1 (Figures 2.6 and 2.4). Figure 2.10 shows that splitting d at the end of the conditional basic blocks makes the interference graph 2-colorable for the first example. Figure 2.11 shows that splitting b and c in the middle of the loop also make the interference graph 2-colorable for the second example, but one has to make sure that the copies are done in parallel to perform a swap of their colors.[5]

The price of splitting variables is that, if the corresponding sub-variables are indeed allocated to different registers, the inserted copies will actually have to be performed with `move` instructions, which degrades performance. But hopefully, this splitting helped to avoid a spill which is usually more expensive than moves.

The most famous example of live-range splitting in compilation is the SSA form, which will be presented in Section 2.3. While splitting helps for coloring, one should

[4] This is of course true only with the relaxed Definition 2.8 of interference.

[5] This is the same problem as sequentializing parallel copies when going out of SSA, see Section 2.3.6.

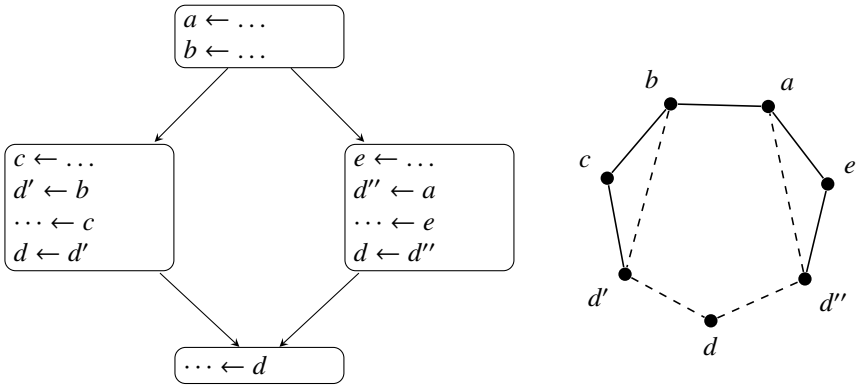


Figure 2.10: Splitting d makes 2 registers enough for the program of Figure 2.6.

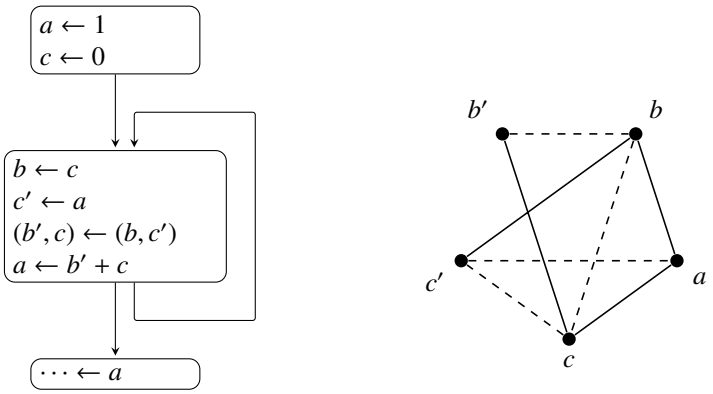


Figure 2.11: Splitting b and c in parallel makes it possible to swap their colors.

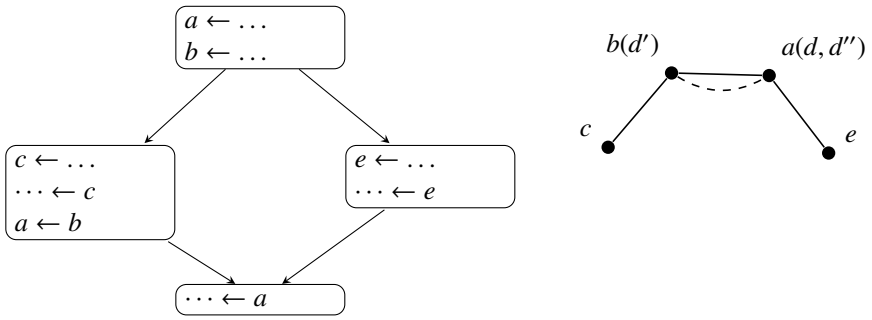


Figure 2.12: Coalescing in the example of Figure 2.10: b with d', a with d, and d''.

keep in mind that whatever the splitting, it will *not* lower Maxlive, hence it cannot solve the problem if $\Omega > R$.

Variable coalescing: Finally, the converse of the splitting technique might help: the *coalescing*. This corresponds to grouping two different non-interfering variables into one, by replacing every occurrence of the second by the first. The effect on the interference graph is that the corresponding nodes are merged, hence decreasing by one the degree of the common neighbors and augmenting their chance to be simplified. This was remarked by George and Appel [1996], and actually used by Vegdahl [1999] to improve the simplification scheme of Chaitin et al.

Of course, the problem is that the node resulting from the merge may have more neighbors. In general, it is hard to know if a particular coalescing will break the colorability of the graph; this will be discussed in Chapter 5. Global or local rules can help deciding if a coalescing is "safe," for instance Briggs's and George's rules use neighborhood criteria to make sure the resulting node will still be simplifiable at some point of Chaitin et al.'s algorithm. Hailperin [2005] does a nice formal model of the power of these two rules. We will heavily discuss coalescing rules in Chapter 6.

In practice, coalescing is often performed only between nodes which have an affinity (see Definition 2.12), so that the corresponding copies can be removed from the program code. As an example, Figure 2.12 shows a possible coalescing of the program in Figure 2.10: three of the four copies are removed by coalescing b with d', a with d'', and d with d''. The fact that common neighbors are more likely to be simplified is then more a nice side effect than the primary goal of the coalescing.

2.2.4 Iterated Register Coalescing (IRC)

Classical approaches for graph-based register allocation integrate in the same framework spilling, coalescing, and coloring, the last one being the final assignment of variables to registers. This is the case in the Iterated Register Coalescing approach proposed by George and Appel [1996], a modified version of the original allocation scheme of Chaitin [1982] and of improvements due to Briggs et al. [1994]. The problem is also modeled with the interference graph of the program, on which the greedy approach of Chaitin et al. is used to try to color the graph with R colors. This involves a combination of the following mechanisms. The execution flow between them is shown graphically on Figure 2.13:

a) build: the interference graph is built from the program;

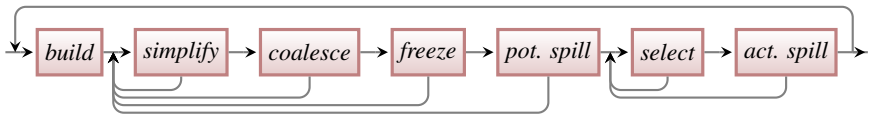


Figure 2.13: Flow diagram of the Iterated Register Coalescing scheme.

b) simplify: a vertex/variable with at most $(R - 1)$ neighbors can be simplified (removed) from the graph since it will be easy to color afterwards (this is the same mechanism as in Function `Is_kGreedy`, page 21). Vertices involved in copy instructions are not simplified to get a chance to be *coalesced*;

c) coalescing: removing a copy instruction can be done by merging the two vertices involved in the move; this is performed in a *conservative* way, i.e., with simple rules that guarantee that the graph remains greedy-k-colorable;

d) freeze: copy instructions are tested several times to improve they chance of being coalesced by the conservative tests. When no more copy has a chance to be coalesced, the algorithm "freezes" one copy, i.e., gives up on this one and will never test it again;

e) potential spill: when all vertices have at least R neighbors, some vertex is simplified and marked as a "potential" spill;

f) select: when the graph is empty, the vertices are colored in the reverse order of their simplification. Each vertex is given a color not used by its already-colored neighbors;

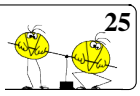
g) actual spill: if no color is available for a vertex marked as a potential spill, an actual spill is performed, i.e., loads and stores are inserted in the code;

h) rebuild: if there was a spill, the interference graph is rebuilt and the coloring procedure is restarted.

Such an approach gives fairly good results. But the main reason for its success is certainly its simplicity both from a conceptual and an implementation point of view. Weights can be easily added to guide the spilling and the coalescing. This allows it to take into account different dynamic execution frequencies of basic blocks. Physical registers can be added as specific "pre-colored" vertices. Register constraints are expressed by adding copies in the code, so that the coalescing elegantly deals with them. "Smarter" coloring strategies for the select phase, such as biased coloring, can be used to improve the coalescing. However, this approach has also several weaknesses for both spilling and coalescing:

- For spilling, once a vertex is actually spilled, there is no obvious method to decide where to place loads and stores, except the simple but inefficient "spill-everywhere" approach, where a `store` is inserted after each definition, and a `load` before each use of the spilled variable.[6] Even worse, it can happen that some spilling is done even if this actually does not help to make the graph k-colorable.

[6] Refinements can be done afterwards, for instance, Chaitin et al. [1981] states that unnecessary loads can be removed using a pass of dead-code elimination.



- For coalescing, although simple and appealing, conservative coalescing is sometimes not aggressive enough and too many moves may remain in the code. Finally, even if live-range splitting is sometimes considered in such a framework, it is very hard to control the interplay between spilling and splitting/coalescing.

In the initial scheme of Chaitin et al., the coalescing was aggressive, i.e., copies were eliminated regardless of their effect on the colorability of the graph. But when Briggs et al. [1994] introduced live-range splitting in this scheme, they moved to a conservative coalescing that would not cancel the effects of the splitting. With the growing difference in speed between accessing memory and accessing registers, this is often better to have more `move` instructions if this saves a spill. This settled conservative coalescing in the IRC scheme, delegating the coalescing of initial program moves to an (optional) constant propagation phase, so that the less powerful coloring-aware coalescing had only to deal with copies inserted by live-range splitting.

2.3 Static Single Assignment form

Static Single Assignment (SSA) form is a property for intermediate representations widely used in modern compilers, usually because it enables or simplifies well-known optimizations. We will first give definitions and properties of SSA form, then explain why it is interesting for register allocation and the particularities of this form with regard to program code.

2.3.1 Definition of SSA

Definition 2.24. *SSA* form: every variable is textually[7] defined exactly once before being used. Given a variable a, def(a) is the instruction that defines a and use(a) is the *set* of the instructions that use a.

Then, under SSA, there is one unique *static* definition, but it is possible to get multiple *dynamic* definitions—for example, if the definition occurs in a loop. This form is illustrated in Figure 2.14. Usually, SSA is considered *with dominance property*, which will be defined in the next section.

A program can be converted to SSA form by renaming multiple definitions of the same variable into subscripted versions of this variable. At join points of the CFG, multiple SSA variables derived from the same original variable must be merged into one SSA variable depending on where the execution path comes from. This is the purpose of the so-called virtual ϕ-functions.

Definition 2.25. A *ϕ-function* is a virtual operation which can be placed only at the beginning of a basic block (at the program point before the first instruction). It takes as many arguments as the number of incoming flow edges, and return the value of its n^{th} argument when the execution path comes from the n^{th} incoming flow edge.

An example of ϕ-function is given on Figure 2.14: c is defined twice, hence is replaced by c_1 and c_2. At the end of the "if…then…else" construct, there is a use of c, whose value depends on which branch of the condition was taken. The ϕ-function inserted acts as a multiplexer by "choosing" c_1 if the path comes from the left, and c_2 if it comes from the right, defining a third variable c_3 which is the one used afterwards.

[7] In the source code of the program.

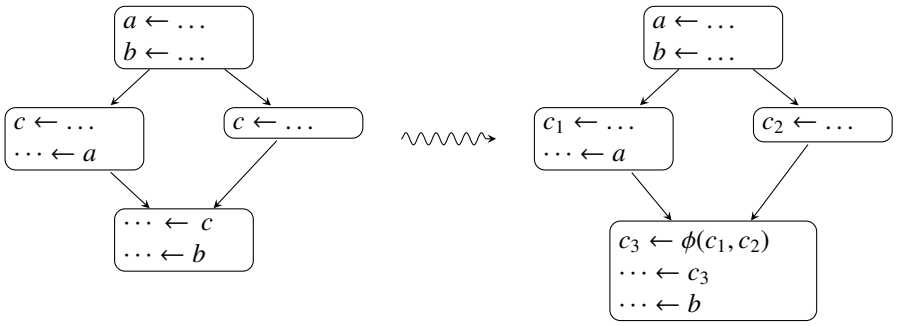


Figure 2.14: A program converted to SSA

Definition 2.26. A program is in *conventional* SSA form (cSSA) if, for all ϕ-functions, all variables involved in the ϕ-function (the arguments and results) can be renamed with a common name.

This means that under cSSA, the arguments and results of a ϕ-function must not interfere. A program converted to SSA form using the method of Cytron et al. [1991] is under cSSA. This property is useful, for instance, if the result of a ϕ-function is spilled, then it ensures that arguments can be spilled to the same memory location. Some optimization can break the conventional property, like copy folding and code motion, but Sreedhar et al. [1999] gave a method to convert SSA back to cSSA.

2.3.2 The dominance property

Definition 2.27. An instruction s—or a block of instructions—*dominates* an other instruction t if every elementary path[8] from the root of the program to t goes through s. The notation is $s \preceq t$.

Definition 2.28. SSA is said to be *with dominance property* if, for every variable a, $\mathrm{def}(a)$ dominates every element of $\mathrm{use}(a)$.

Lemma 2.29. *If SSA is with dominance property, for every variable a,* $\mathrm{def}(a)$ *dominates every element of* $\mathit{Live}(a)$.

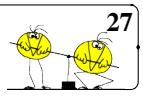
Proof. Suppose that a is live at program point p. Then, there exists a path from p to an instruction u that uses a which does not go through $\mathrm{def}(a)$. If p was not dominated by $\mathrm{def}(a)$, there would exist a path $r \rightsquigarrow p$ from the root of the program to p which does not go through $\mathrm{def}(a)$, hence there would exist a path $r \rightsquigarrow p \rightsquigarrow u$ with the same property, which contradicts the dominance property. □

The following theorem is well-known and we will need it for some later proofs, so we recall it here for completeness.

Theorem 2.30. *Dominance is a partial order: it is antisymmetric, reflexive and transitive.*

Proof. Proof of the three properties:

[8] Path which contains at most one time any instruction.



- antisymmetric: we suppose $s \preceq t$ and $t \preceq s$. Let us consider an elementary path $r \rightsquigarrow t$ from r to t. This path goes through s since $s \preceq t$: it can be split into $r \rightsquigarrow s \rightsquigarrow t$. Again, since $t \preceq s$, the path from the root to s goes through t and the initial path writes $r \rightsquigarrow t \rightsquigarrow s \rightsquigarrow t$. But the first path was an elementary path hence the only possibility is that $t \rightsquigarrow s \rightsquigarrow t$ is of zero length, i.e., $s = t$;

- reflexive: s is the last element of any path from the root to itself, hence it dominates itself;

- transitive: if $s \preceq t$ and $t \preceq u$. Let $r \rightsquigarrow u$ be a path from the root to u. It contains t and can be written $r \rightsquigarrow t \rightsquigarrow u$. But $s \preceq t$ so one can split $r \rightsquigarrow t$ in $r \rightsquigarrow s \rightsquigarrow t$ which means that $r \rightsquigarrow u$ contains s and $s \preceq u$.

□

Definition 2.31. The *dominance graph* is the Hasse diagram[9] of the graph where the vertices are the instructions and the (directed) edges indicate dominance:

$$s \to t \iff s \preceq t$$

Property 2.32. *If s and t dominate u, then either s dominates t, or t dominates s.*

Proof. Consider an elementary path from the root r to u. This path contains s and t by definition. Without loss of generality, one can suppose that s appears before t on this path: $r \rightsquigarrow s \rightsquigarrow t \rightsquigarrow u$. Suppose there exists a path $r \rightsquigarrow^* t$ from r to t which does not go through s, then one could extend this path to u, and $r \rightsquigarrow^* t \rightsquigarrow u$ would be a path from the root to u not going through s which contradicts the fact that $s \preceq u$. Hence every path from r to t goes through s and $s \preceq t$. □

Note on live-ranges under SSA. The dominance property seems to be contradictory with the existence of ϕ-functions. Indeed, when looking at the example of Figure 2.14, the use of c_1 and c_2 do *not* seem dominated by their definition. But in fact, the ϕ-function is not a normal instruction and its semantics is that assignments are performed "somewhere" on the incoming edges. This means that c_1 and c_2 are in fact not *live_in* of the basic block where the ϕ-function is. The converse is true for the definition: c_3 is not *live_out* of the preceding basic blocks. Hence, the live-range of a variable used in a ϕ-function under SSA *ends* at the end of the basic block preceding the ϕ-function (unless it is still used after the ϕ-function), and the live-range of a variable defined by a ϕ-function *starts* at the beginning of the basic block where the ϕ-function is.

The live-ranges are important for the shape of the interference graph. Indeed, it is important that c_1 and c_2 do not interfere (if the ϕ-function is their last use) because that is the purpose of splitting under SSA. Interferences exists whenever live-ranges intersect, and in the following we prefer to stick to this definition and not take value into account. This is in fact not a limitation under SSA since variables are defined only once, hence, if at its definition a variable is defined as $[b \leftarrow a]$, b will always have the same value as a and can be renamed at every of its uses. This can be easily done with a copy folding algorithm, a very common optimization in compilers.[10]

[9] Graph whith the transitive edges removed.

[10] However, this breaks the "conventional" property of cSSA codes.

2.3.3 Properties of SSA

Property 2.33. *Under SSA, if a interferes with b, then either* $\text{def}(a) \preceq \text{def}(b)$*, or* $\text{def}(b) \preceq \text{def}(a)$.

Proof. Let p be a program point where a and b are simultaneously alive. Using Lemma 2.29, we know that $\text{def}(a) \preceq p$ and $\text{def}(b) \preceq p$. The property 2.32 concludes. □

Corollary 2.34. *Under SSA, if a interferes with b and* $\text{def}(a) \preceq \text{def}(b)$*, then* $\text{def}(b) \in \text{live}(a)$.

Proof. $\text{def}(a) \preceq \text{def}(b)$, and since a and b interfere, by Definition 2.8 of the interference, a is alive at $\text{def}(b)$, which means $\text{def}(b) \in \text{live}(a)$. □

Theorem 2.35. *The dominance graph under SSA is a tree.*

Proof. A vertex u has only one direct predecessor: if s and t dominate u, then Property 2.32 states that one of the two dominates the other. For instance, $s \preceq t$, then $s \to u$ is a transitive edge and does not appear in the Hasse diagram. Moreover the graph is connected since the root dominates every vertex. □

Corollary 2.36. *Under SSA, the live-ranges are subtrees of the dominance graph.*

Proof. Let us consider a variable a and a point p of its live-range. Let us consider the shortest path P_{dom} from the definition of a to p on the dominance tree. Since a is alive at p there is a path P_{CFG}, *on the CFG*, from p to a use of a which does not go through $\text{def}(a)$.

For any point $p' \neq \text{def}(a)$ of path P_{dom}, $\text{def}(a) \preceq p' \preceq p$ because $\text{def}(a) \preceq p$. Hence, there is also a path P'_{CFG}, on the CFG, from p' to p. Moreover, one can choose P'_{CFG} so that it does not go through $\text{def}(a)$, else this would mean $p' \preceq \text{def}(a)$ which is impossible because of the antisymmetric property of the dominance. Hence, the concatenation of P'_{CFG} with P_{CFG} is a path, on the CFG, from p' to a use of a which does not go through the definition of a. This means a is alive at p'.

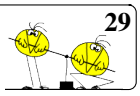
Hence, every shortest path P_{dom}, on the dominance tree, from $\text{def}(a)$ to a point where a is alive contains only points where a is alive. They are sub-paths of the dominance tree, and since have a point in common—the definition of a—, the union of these paths is connected: it is a subtree of the dominance graph. □

Figure 2.15 shows the live-ranges of the previous SSA example, Figure 2.14. The conditional branches do not dominate the last basic block, so the dominance graph is a tree and the SSA live-ranges are subtrees of this tree.

2.3.4 SSA interference graph is chordal

In 2005, we discovered that, under SSA, the interference graph of a program is chordal. Independently, Brisk et al. [2005]; Pereira and Palsberg [2005], and Hack et al. [2006] made the same observation. Note that the interference graph depends on the interference notion. We use Definition 2.8 for that purpose,[11] and the shape of the live-ranges under SSA are explained in the note page 28.

[11] Note that, under SSA, there is only one definition, hence, if variable b is defined as $[b \leftarrow a]$, it is possible to replace every occurrence of b by a since they will always be equal. Note that this is akin to aggressive coalescing but is safe to do under SSA as it will not increase the coloring number. Constant propagation can do this efficiently in a first pass, then it is not worth considering the interference definition of Chaitin (Definition 2.11) as there is no remaining copy in the program.



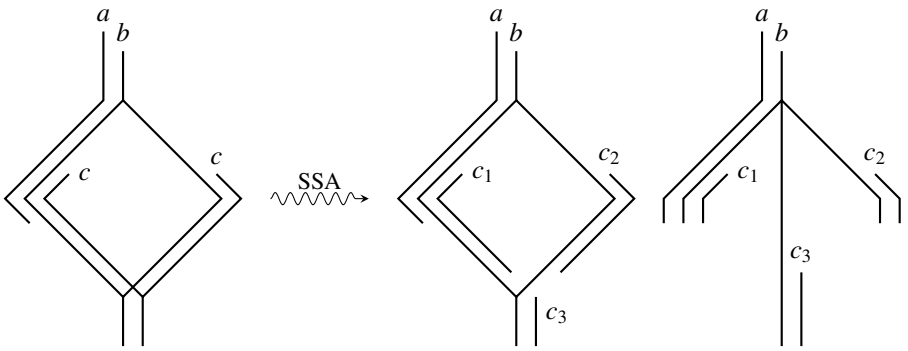


Figure 2.15: Live-ranges of the program of Figure 2.14 and its version under SSA. c_1 and c_2 stop at the end of their definition blocks, while c_3 starts at the beginning of the block with the ϕ-function (see note page 28), hence they are subtrees of the dominance tree.

Theorem 2.37. *The interference graph G of a program under SSA with dominance property is chordal.*

Short proof. Corollary 2.36 states that under SSA, live-ranges are subtrees of the dominance graph. Hence the interference graph is the intersection graph of a family of subtrees (the live-ranges) of a tree (the dominance tree), which is another characterization of chordal graphs [Golumbic, 1980, Thm. 4.8]. □

It is possible to give a more direct proof, without using the characterization of chordal graphs as the intersection graph of subtrees of a tree. It was by finding this proof that we first realized that the interference graph of a program under SSA is chordal, which is the very first contribution of this thesis. Then, we figured out that the chordal representation as subtrees of a tree would perfectly match the live-ranges under SSA.

Proof. Let G be an interference graph of a program under SSA. Let us define the following orientation of the edges: if $\text{def}(u) \preceq \text{def}(v)$, then $u \to v$. Property 2.33 states that every edge is directed. Consider a cycle C of length at least 4 in G, if there is one. From Theorem 2.30, the dominance relation is a partial order: C cannot form a directed cycle, thus there are two edges $u \to v$ and $v \leftarrow w$, directed from u to v and from w to v, i.e., the definitions of u and of w dominate the definition of v. Since u and v interfere, and $u \to v$, u is alive at $\text{def}(v)$ and the same is true for w. u and w are both alive at the $\text{def}(v)$, they interfere and there is an edge between u and w in the graph, i.e., a chord in C.

□

As a chordal graph, the interference graph of a program under SSA is perfect, hence $\chi(G) = \omega(G)$: the coloring number is equal to the size of the largest clique. We will now see how to correlate $\chi(G)$ with Maxlive, the maximum number of simultaneously alive variables. Before seeing it, we need the following property which links together the number of alive variables in the program and the cliques of the interference graph. The size of these cliques fixes the number of colors required.

Property 2.38. *In an interference graph G under SSA:*

- *for any program point, the set of live variables form a clique in G;*
- *reciprocally, to every clique of G of size ω corresponds a program point where at least ω variables are alive.*

Proof. The first point is obvious since variables simultaneously alive form a clique (remember we still use the relaxed Definition 2.8 of interference). Now, consider a clique in G with directed edges as in the proof of Theorem 2.37. Since there is no directed cycle, there is a vertex u in the clique such that, for any other vertex v in the clique, (u, v) is directed from v to u, i.e., the definition of u is dominated by the definition of any other vertex. Thus all variables in the clique are live at the definition of u, which proves the second point. □

Corollary 2.39. *Under SSA, the coloring number of the interference graph is Maxlive, i.e., $\chi(G) = \Omega$.*

Proof. Chordal graphs are perfect graphs, hence their coloring number $\chi(G)$ equals their clique number $\omega(G)$ [Golumbic, 1980]. From Property 2.38, the largest clique is of size Maxlive, hence the interference graph of a program under SSA is Ω-colorable. □

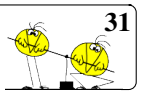
Back to register allocation. In Section 2.2.2.6, we proved Property 2.23, which states that a k-chordal graph is also greedy-k-colorable. Now that we know that the interference graph of a program under SSA is chordal, the consequence of this basic property, to our knowledge not mentioned in the compiler literature before our work, is particularly interesting for register allocation in the context of SSA. In Property 2.23, we just used the well-known proof that a simplicial elimination scheme leads to an optimal coloring for a chordal graph, as recalled by Pereira and Palsberg [2005]. But our definition of greedy-k-colorability implies more. In register allocation, the number of registers R is fixed and there is, in general, no point in trying to use as few registers as possible: just fewer than R is sufficient. In other words, it is possible to use an optimal on-line coloring such as a simplicial scheme or a smallest last order, but, as the number of registers R is known, it is also possible to simply use any Chaitin-like simplification scheme, i.e., to remove vertices with degree less than R in any order.

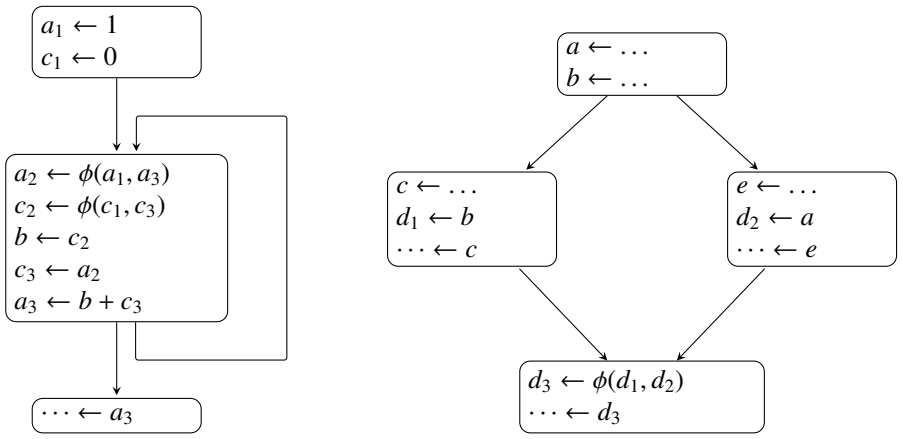
Moreover, using Corollary 2.39, we know that, if $\Omega \leq R$, there is no need to spill and the greedy coloring scheme of Chaitin et al. will manage to color the interference graph with R colors. This implies that, under SSA we have an exact test to decide if some spilling is required or not.

Finally, we mentioned in Section 2.2.2.4 that the representation of chordal graphs as subtrees of a tree makes it possible to color them by *scanning* the tree from the root to the leave. In the context of SSA form, this is directly applicable by scanning the program from the root to the leaves of the dominance tree, assigning colors to the live-ranges when encountered.

2.3.5 Why is coloring polynomial under SSA?

In the general case, it is NP-complete to decide if R registers are enough, while under SSA, the interference graph is chordal hence the same problem is polynomial. Why? Because SSA *splits* variables by using ϕ-functions. Indeed, we have seen in





(a) Program of Fig. 2.4 (b) Program of Fig. 2.6

Figure 2.16: Running examples under SSA

Section 2.2.3 that splitting variables helps. This was observed by Fabri [1979] who explained that splitting variables can lower the clique number of the interference graph to Maxlive. Under SSA, variables are split at *the only necessary points*, at the dominance frontier, i.e., at points where there are cycles in the live-ranges on the dominance tree: ϕ-functions splits the live-ranges which spawn across branches without following paths on the tree, so that they become disconnected subtrees.

Note that additional splitting points *cannot lower* the clique number below Maxlive; however, additional splitting can still be interesting if one looks for split points where placing `move` instructions is cheaper. For instance, an instruction inside a loop is usually dynamically more expensive than an instruction outside the loop. Hence, even if more splitting is unnecessary in terms of coloring, it might still be interesting in terms of coalescing, i.e., for minimizing the number of (weighted) copy instructions.

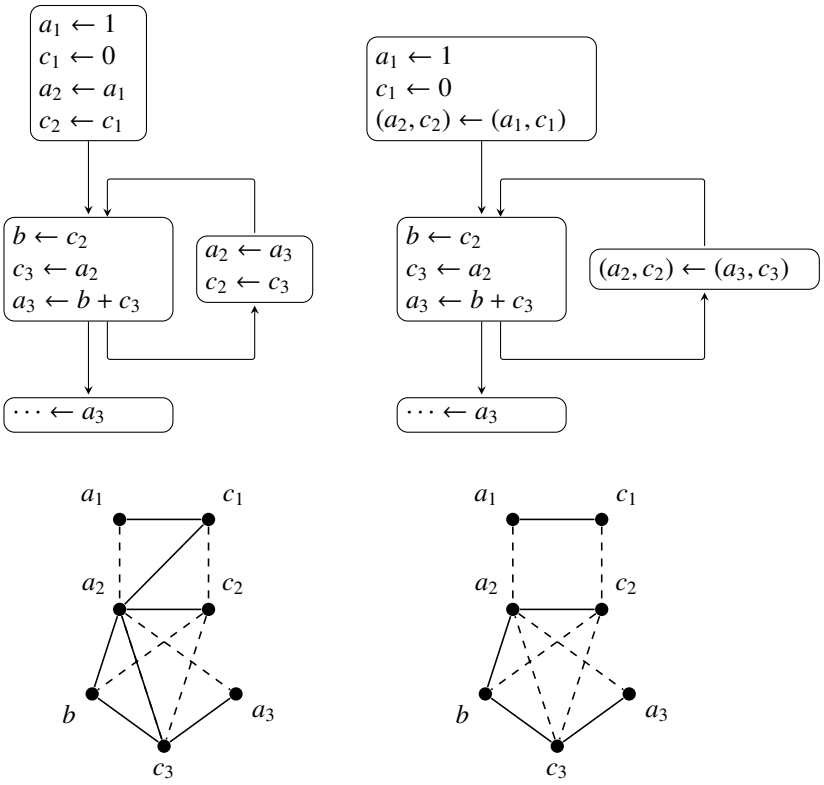
Finally, if SSA provides a tool for splitting variables efficiently, it is certainly not the unique way to split variables so that the remaining interference graph gets chordal. For instance, one of the first examples of this chapter, Figure 2.4, was split by SSA and presented in Figure 2.16a, but we first gave another example of splitting on Figure 2.11. Both splitting made the interference graph 2-colorable while the non-split initial program needed 3 registers. Figure 2.16b also shows the SSA splitting of the running example of the odd-length cycle (initial program on Figure 2.6).

We will see in the next chapter, Section 3.1.2, a more detailed explanation on the effects of splitting on the complexity of register allocation.

2.3.6 SSA form is not machine code

SSA form is not machine code: ϕ-functions are virtual instructions which do not exist in hardware. Even though, a ϕ-function represents a transfer of values between variables, and whenever the source variables are different from the destination variables—they can be equal, for instance after some coalescing—, ϕ-functions need to be materialized by adding `move` instructions "on" the incoming edges. This problem in known as "going out of SSA" in the literature.

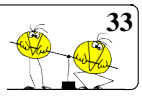
Two problems arise: First, an edge cannot contain any code. One possibility is to place the copies at the end of the source basic block, but if this block has more than



(a) Sequential copies

(b) Parallel copies

Figure 2.17: Sequentializing copies creates new interferences.



one successor, the copies will still be executed even if another edge is chosen when the program is run. Second, if multiple copies must be added to an edge, the order in which they are sequentialized is important—for instance, if some variables are used both as argument and as result of ϕ-functions.

Critical edges: The first problem is related to the notion of critical edges.

Definition 2.40. A *critical edge* is an edge in the CFG which goes from a basic block with more than one successor to a basic block with more that one predecessor.

For instance, the back edge of a loop can be a critical edge, depending on whether in ends in a jump or a branch. Structured loop constructs tend to produce a branch at the bottom of the loop, which produces a critical edge (see for instance Figure 2.16a). If some code must be placed on a critical edge, it is dangerous to place it at the end of the preceding basic block—the code will still be executed if one of the other leaving edges is chosen—or at the beginning of the following basic block—the code will still be executed if the path comes from one of the other incoming edges.

There are (at least) two solutions to this problem. One possibility is to still place code at the borders of basic blocks, but to make sure it does not modify the semantic of the program when other edges are chosen. For instance, this code should not re-define a variable used later on another execution path. Sreedhar et al. [1999] chose to add copies to new variables for the arguments on the preceding blocks, *and* also a copy for each variable defined by a ϕ-function. This approach is explained in Chapter 7, Section 7.1.

Another solution is to *split* the critical edge and add the code to the newly created basic block.

Definition 2.41. *Splitting* an edge going from a source basic block B_s to a destination basic block B_d is done by creating a new basic block B_n, deleting the edge and creating two new edges from B_s to B_n and B_n to B_d.

Parallel ϕ-functions: The second problem is related to the fact that the semantics of multiple ϕ-functions at the beginning of a basic block is that they are executed in *parallel*. It is a mistake to consider them as sequential instructions, for two reasons: First, if a variable is both used and defined in two ϕ-functions, one might erase the value before having used it. Here is an example:

$$\begin{aligned} a &\leftarrow \phi(\dots) \\ \dots &\leftarrow \phi(a, \dots) \end{aligned}$$

Second, if sequentialized, interferences which did not exist beforehand are created: interferences between arguments and results of the ϕ-functions. For instance, the two codes of Figure 2.17 have been obtained from Figure 2.16a by replacing the ϕ-functions with `move` instructions: on the first basic block, and on a new basic block on the back edge of the loop. On Figure (a), these instructions copies are sequentialized which creates interferences between c_1 and a_2, and between a_2 and c_3. The interference graph is then 3-colorable. On Figure (b), parallel copies are used which keeps the graph 2-colorable as in the is the original SSA code.

Hence the parallel semantics of ϕ-functions is crucial and should be kept for as long as possible. Of course, in the end, the final code will be sequential, but it is better to use instructions with parallel semantics until the very last moment so that one does

not lose some information on the interference graph. In order not to forget the parallel semantics of ϕ-functions, Hack [2007] proposes to replace multiple ϕ-functions by one ϕ-function using a matrix notation:

$$\begin{array}{lcl} x & \leftarrow & \phi(x_1, x_2, \ldots, x_n) \\ y & \leftarrow & \phi(y_1, y_2, \ldots, y_n) \\ & \vdots & \\ z & \leftarrow & \phi(z_1, z_2, \ldots, z_n) \end{array} \quad \rightsquigarrow \quad \begin{pmatrix} x \\ y \\ \vdots \\ z \end{pmatrix} \leftarrow \phi \begin{pmatrix} x_1 & x_2 & \ldots & x_n \\ y_1 & y_2 & \ldots & y_n \\ \vdots & \vdots & \ddots & \vdots \\ z_1 & z_2 & \ldots & z_n \end{pmatrix}$$

This means that if arriving, for instance, from the second incoming edge, all copies $[x \leftarrow x_2], [y \leftarrow y_2], \ldots, [z \leftarrow z_2]$ need to be performed *at the same time*. Note however that multiple ϕ-functions at the beginning of the same basic block are *not the only cause* of parallel copies creation. Two ϕ-functions in different basic blocks can also create copies that should be parallel. This is the case for instance if a basic block B has two successors that contains respectively $a \leftarrow \phi(b, \ldots)$ and $b \leftarrow (a, \ldots)$. If, when going out-of-SSA, the copies are added at the end of B, the two moves $[a \leftarrow b]$ and $[b \leftarrow a]$ must obviously be made parallel. The use of the matrix notation should not make us forget that point.

In the end, it will usually be necessary to sequentialize the parallel copies. This is possible without adding more code unless the copies represent a permutation of the variables. In this case, swap instructions or temporary variables can be used for example. We will not go deeper into details here since sequentializing copies will be discussed Chapter 7, Section 7.3.4. We will nevertheless give here a classical example of dead lock: the swap of two values.

$$\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \phi \begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

In this case, the swap between values of a and b coming from the second edge cannot be sequentialized as the first copy executed would overwrite the value needed for the second. In the absence of swap instructions in the architecture,[12] another free register needs to be used as a temporary value holder and the instructions $[t \leftarrow a; a \leftarrow b; b \leftarrow t;]$ are performed. This is problematic if the register pressure at this point is already equal to R, in which case a spill is needed.

To prevent the creation of artificial interferences too early, it is best to represent copies due to ϕ-functions using parallel copies—$(a, b) \leftarrow (b, a)$ in our example—which we introduce in the next section.

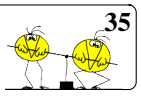
2.3.7 Splitting and parallel copies

Definition 2.42. A *parallel copy* is a virtual instruction taking n arguments and defining n variables, *simultaneously*, from these arguments. The notation is

$$(v_1, v_2, \ldots, v_n) \leftarrow (a_1, a_2, \ldots, a_n)$$

This is a fundamental instruction when dealing with program splitting. To split all variables at one program point, one needs to duplicate all variables alive at this point, and insert a parallel copy between all the variables and their duplicates. Trying to split by inserting normal, i.e., sequentialized copies, would create interferences between

[12] Or the possibility to emulate them, for instance by using three XOR.



some variables and the duplicates of others. Parallel copies can be seen as a way to "reorganize" values in variables, and are sometimes referred to as "shuffle code."

However, there is no such hardware instruction. At best, one can find instructions to swap values in registers or perform up to a fixed number of copies in parallel, e.g., four copies on a 4-way VLIW architecture, or emulate a swap by using three consecutive XOR. In the end, parallel copies will need to be instantiated with actual machine code. As said previously, these matters are discussed in Chapter 7.

2.4 Conclusion

In this chapter, we defined notations and objects that we will manipulate in the next chapters. We provided background information on register allocation, defining programs, live-ranges and interference graphs. We discussed the colorability of the interference graph with regards to Maxlive, the maximum number of variables simultaneously alive, and introduced interesting graphs structures for interference graphs. Then, we discussed possibilities of modifying the program whenever R registers are not sufficient to color the graph. Finally, we introduced the SSA form, which is concerned by two results of ours: first, the interference graph of a program under SSA is *chordal*; second, chordal graphs can be colored using the simple greedy algorithm of Chaitin et al. [1981]: they are greedy-colorable, a property that we introduced in this chapter.

Please keep in mind that programs are always considered strict, and SSA programs are considered with dominance property. Also, the notion of interference if very important for the shape of the interference graph, in particular, for any program point there is a clique in the graph only if variables alive at the same time interfere *even if* they have the same value. Finally, we will always consider that we have only one type of registers during this thesis. In practice, different classes exist, like integer and boolean registers. For disjoint classes, they can be considered independently, but particularities like register aliasing complicates the problem. These subtleties will be discussed in conclusion, i.e., Chapter 8, along with more practical advices.

Using these grounds, we will build a new way of viewing register allocation in two phases in the next chapters: first, spilling with some splitting, second, coloring with some coalescing.

PROOF, n. Evidence having a shade more of plausibility than of unlikelihood. The testimony of two credible witnesses as opposed to that of only one.

Ambrose Bierce (1842 – 1914), *The Devil's Dictionary*

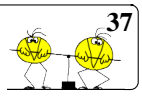
3

What does the NP-completeness proof of Chaitin et al. really prove?

The goal of register allocation is to map the variables of a program into physical memory locations (main memory or machine registers). Accessing a register is usually faster than accessing memory, thus one tries to use registers as much as possible. When this is not possible, some variables must be transferred, "spilled," to and from memory. This has a cost, the cost of the load and store operations, which should be avoided as much as possible. Solving this problem has been a necessity since the very first compilers. And, although it is very simple to state, many efforts have been made to find the best possible solutions, as the problem is in practice quite complicated. We will continue the introduction of this chapter with a long but hopefully meaningful explanation of why people usually view register allocation as a difficult problem, why this is not always true, and why we propose to study, again, the complexity of this problem.

Classical approaches for register allocation are based on fast graph coloring algorithms. A widely-used algorithm is the Iterated Register Coalescing (IRC) proposed by George and Appel [1996], a modified version of previous developments by Chaitin et al. [1981]; Chaitin [1982], and Briggs et al. [1994]. In these heuristics, *spilling*, *coalescing* (i.e., removing register-to-register moves), and *coloring* (i.e., assigning variables to registers) are done in the same framework. Priorities among these transformations are done implicitly with cost functions. *Splitting* (adding register-to-register moves) can also be integrated in this framework. Such techniques are well-established and used in optimizing compilers. However, there are several reasons to revisit these approaches and register allocation in general. First, some algorithms not considered in the past, because they were too time-consuming, can be good candidates today: processors used for compilation are now much faster and, for critical applications, industrial compilers are also ready to accept longer compilation times. Second, the increasing difference on most architectures between the cost of a memory access and the cost of a register access suggests to focus on heuristics that give more importance to spilling cost minimization, possibly at the price of additional register-to-register moves. Finally, there are many pitfalls and folk theorems concerning the complexity of the register allocation problem that are worth clarifying.

This last point is particularly interesting to note. Chaitin et al. [1981] modeled the problem of allocating variables of a program to R registers as the problem of coloring, with R colors, the corresponding interference graph. By showing that any graph is the interference graph of a program, and because Graph k-Colorability is NP-complete [Garey and Johnson, 1979, Problem GT4], they proved that, in their model, deciding if R registers are sufficient to perform register allocation without any spill is NP-complete.



And from this date up to now, heuristics have been used for spilling, coalescing, splitting, coloring, etc. As a consequence, the argument "register allocation *is* graph coloring, therefore it is NP-complete" is one of the first statements of many papers on register allocation. The following quote comes from the introduction of an article by Konstantinos Sagonas and Erik Stenman [2003], but many others can be found:

> "In this case [global register allocation], control-flow enters the picture and obtaining an optimal mapping becomes an NP-complete problem [...]"

This is *not* what Chaitin et al. proved. Actually, going from register allocation to graph coloring is just a way of modeling the problem, not an equivalence. In particular, this model does not take into account the fact that a variable can be moved from a register to another using live-range splitting. Our impression was that there is a misunderstanding of the implications of Chaitin et al.'s proof in the community. While it is true that most problems related to register allocation are NP-complete, identifying register allocation to graph coloring can make us forget what Chaitin et al.'s proof actually shows. In particular, it is commonly believed that, in absence of instruction rescheduling, it is NP-complete to decide if the program variables can be allocated to R registers with no spilling, even if live-range splitting is allowed.

Until very recently, only a few authors addressed the complexity of register allocation in more details. Maybe the most interesting complexity results are those of Liberatore et al. [1999]; Farach-Colton and Liberatore [2000], who analyze the reasons why *optimal* spilling is hard for basic blocks. In this case, the coloring phase is of course easy because, after some variable renaming, the interference graph is an interval graph, but deciding *which* variables to spill and *where* to spill them is in general difficult. They call this phase "allocation," as it decides which variables are allocated in memory and which are allocated in registers, and differentiate it from the second phase, called "register assignment." In this phase, variables are mapped to registers, possibly removing move instructions by coalescing, or introducing move instructions by splitting. When loads and stores are more expensive than moves, such an approach is worth exploring. It was experimented by Appel and George [2001] and also advocated by Knobe and Zadeck [1992]; Hack et al. [2006].

The last example clearly states that, for a basic block, the problem lies in the spilling, not the coloring. More recently, we discovered that, under Static Single Assignment (SSA) form, the interference graph of a program is chordal (see Theorem 2.37). Brisk et al. [2005]; Pereira and Palsberg [2005], and Hack et al. [2006] independently made the same observation. This theorems shows it is *easy* to decide if R registers are sufficient for a program under SSA. How come the SSA case is not covered by Chaitin et al.'s proof? Combined with the idea of spilling before coloring so that Maxlive $\leq R$, this led Pereira and Palsberg [2006] to wonder where the NP-completeness of Chaitin et al.'s proof (apparently) disappeared:

> "Can we do polynomial-time register allocation by first transforming the program to SSA form, then doing linear-time register allocation for the SSA form, and finally doing SSA elimination while maintaining the mapping from temporaries to registers?"

All this needs to be done when Maxlive $\leq R$ of course, otherwise some spilling is necessary. They show that, if register swaps are not available, the answer is "no" unless P=NP. The NP-completeness proof of Pereira and Palsberg is interesting, but we feel it does not completely explain why register allocation is difficult. Basically, it shows that

if we decide *a priori* what the splitting points are, i.e., where register-to-register moves can be placed (in their case, the splitting points are defined by the ϕ-functions), then it is NP-complete to choose the right colors.[1] However, there is no reason to restrict to splitting only at points given by SSA. Actually, we will show that, when we can choose the splitting points, when we are free to add program blocks to remove critical edges (the standard *edge splitting* technique), then it is easy, except for a few particular cases, to decide if and how we can assign variables to registers without spilling.

Hence, to answer the question: "Where did the complexity disappear?" a good lead would be that splitting variables simplifies the problem. And SSA splits variables with multiple definitions (see Section 2.3.1). Fabri [1979] already observed, working on allocation of arrays into memory, that splitting could reduce the chromatic number down to Maxlive. Of course, splitting has a cost, but only the cost of a move instruction, which is often better than a spill. So the introduction of splitting raises some questions:

- When is Chaitin et al.'s proof applicable?
- What are the limits of Chaitin et al.'s proof?
- How far can Chaitin et al.'s proof be extended to cover other cases?

This chapter acts as a second introduction to this thesis. We will present here the preliminary work that led us to revisit register allocation. In a first part, we will stress Chaitin et al.'s proof on its weakest points, by successively patching the proof and pointing to newly created weak spots. We tried to do it didactically, starting from the original proof and acting as would act someone skeptical, constantly finding new points to argue.[2] Then we will illustrate the limits of this proof by showing two configurations under which the problem of knowing whether R registers are sufficient or not becomes easy—one of them being the SSA form. At this point, we will define what we think is responsible for the complexity of this problem: the "multiplexing regions." Whenever there is no critical edge in these multiplexing regions, we will promote the practical way of doing register allocation in two phases in place of the classical graph-based algorithm in one phase. First phase: spill variables so that Maxlive becomes less that R, and split variables so that the graph becomes R-colorable. Second phase, color while performing coalescing to reduce the effects of having split the program. Finally, we will come back to the outline of this thesis and explain how the next chapters flow from this one.

3.1 NP-completeness proofs

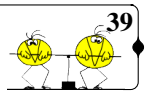
In this section, we will present variations of the NP-completeness proof of Chaitin et al. [1981], to show how much modification of the problem it can endure, and what it cannot.

3.1.1 Direct consequences of Chaitin et al.'s proof

Let us examine Chaitin et al.'s NP-completeness proof, a proof by reduction from Graph k-Colorability [Garey and Johnson, 1979, Problem GT4].

[1] Note that their proof forbid the use of register swaps, while for instance Hack et al. [2006], who actually perform register allocation under SSA, consider they do have them.

[2] For example: – Right,... but *what if* I am allowed to...?



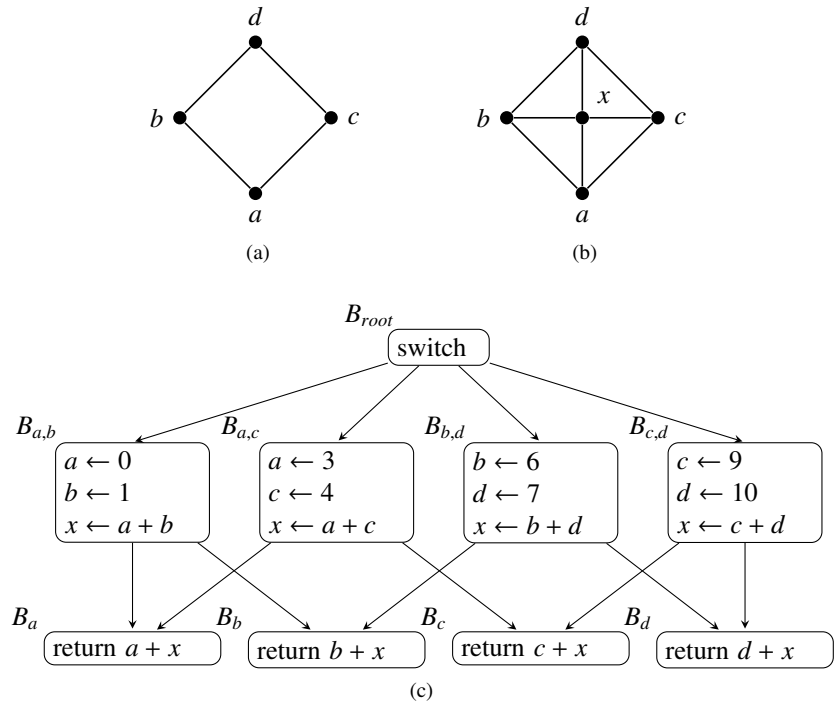


Figure 3.1: Chaitin et al.'s reduction: program (c) built from a cycle of length 4 (a) and its interference graph (b).

Problem: GRAPH k-COLORABILITY
Instance. An undirected graph $G = (V, E)$ and an integer k.
Question. Is it possible to color the graph with k colors, i.e., is there a color $c(v)$ in $\{1, \ldots, k\}$, for each vertex $v \in V$, such that $c(v) \neq c(u)$ for each edge $(u, v) \in E$?

This problem is well-known to be NP-complete if G is arbitrary, even for a fixed $k \geq 3$.

For the reduction, Chaitin et al. [1981] build a program with $|V| + 1$ variables, one for each vertex $u \in V$ and an additional variable x, as shown on Figure 3.1. For each (u, v) in E, a block $B_{u,v}$ defines u, v, and x. For each $u \in V$, a block B_u reads u and x, and returns a new value. Each block $B_{u,v}$ is a direct predecessor in the control-flow graph of B_u and B_v. An entry block switches to all blocks $B_{u,v}$. For G cycle of length 4, on Figure 3.1a, the program is given on 3.1c, and its interference graph is on 3.1b. This is the same graph as G plus a vertex for the variable x, connected to any other vertex; thus x must use an extra color. As such, G is R-colorable if and only if (iff) each variable can be assigned to a unique register for a total of at most $R + 1$ registers. This is what Chaitin et al. proved: for such programs, deciding if one can assign the variables, *this way*, to $R \geq 4$ registers is NP-complete.

What do we mean by "this way?" It means that assigning variables to registers as if coloring the vertices of the interference graph is NP-complete. This is not the only way of coloring variables, which are not atomic, localized objects as the vertices of a graph are: a variable has a life that starts at its definition and can last for a long time. Why should a variable be forced to reside in only one place all along its life? This introduces

the problem of "splitting" variables, which we will explain in the next section.

3.1.2 Splitting variables in Chaitin et al.'s proof

We recalled in Chapter 2 the possibility of *splitting* the live-range of a variable. Basically, introducing a copy creates two new live-ranges in place of the first one, which can be assigned to different places. Chaitin et al.'s proof, at least in its original interpretation, does not address this possibility. Each vertex of the interference graph represents the complete live-range as an atomic object that must always reside in the same register. Furthermore, the fact that the register allocation problem is modeled through the interference graph loses information on the program itself and the exact location of interferences. This is a well-known fact, which led to the development of many different register allocation heuristics, with early development by Chow and Hennessy [1984] and later by Callahan and Koblenz [1991]; Cooper and Simpson [1998], or Lueh et al. [2000], but with no corresponding complexity study even though their situations are *not* covered by the NP-completeness proof of Chaitin et al.

This raises the question: What if splitting live-ranges is allowed? We suppose that it is possible to insert as many copies as we want, anywhere on any basic block. The following theorem proves it does not make the problem any easier.

Theorem 3.1. *It is still NP-complete to decide if R register are enough for a program even if variable splitting is allowed (on basic blocks).*

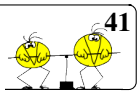
Proof. Let a be a node in the initial graph, i.e., a variable different than x in the program. The key in Chaitin et al.'s proof was that a must reside in the same register in blocks B_a and all blocks $B_{a,y}$ for y neighbor of a in the graph.

Let us consider one neighbor y of a and suppose that the live-range of a has been split. Whatever the splitting, the value of a resides in some variable on the edges going to block B_a since this block needs the value of a. Let us call a_y the variable holding the value of a on the edge from $B_{a,y}$ to B_a. This means that, somewhere inside $B_{a,y}$, there is a copy $[a_y \leftarrow a]$ or $[a_y \leftarrow a_i]$ where a_i is a split variable of a (unless $a_y = a$, i.e., a is not split on this block). Conversely, there should be a copy $[a_j \leftarrow a_y]$ somewhere on B_a, before the use of a. But suppose a has another neighbor y',[3] then there are two copies $[a_i \leftarrow a_y]$ and $[a_j \leftarrow a_{y'}]$ acting concurrently on B_a: the one executed last will finally sets the value of a. Hence the resulting code is false, unless both a_y and $a_{y'}$ share the same register. This is possible, since they are never dynamically alive at the same time: they do not interfere. But if two variables are restricted to be in the same register, it is strictly equivalent to replace them with a common name: all a_y for y neighbor of a can be replaced by a common variable, say a'.

This reasoning, when applied to all variables of the program, shows that, for all nodes v, and whatever the splitting, it the value of the corresponding variable must reside on the same variable v' on all edges going to basic block B_v. For the same reason, variable x must also reside on the same variable x' on all edges going to return blocks.

Consider now all primed variables. Their interference graph is the same as the original program without splitting, and is a subgraph of the interference graph of the split program. Hence, finding a coloring of the split interference graph would give a solution to the initial problem, and conversely, a coloring of the initial program can be

[3] If every node in the graph has only one neighbor, there is no coloring problem.



extended to a solution to the split program (give to all split variables of v the same color as v).

Therefore, the problem remains NP-complete and Chaitin et al.'s proof holds even if live-range splitting is allowed. □

Why did splitting live-ranges not help here? This is because the control-flow edges from $B_{u,v}$ to B_u are *critical* edges, i.e., they go from a block with more than one successor to a block with more than one predecessor. Hence, placing code at the source of the edge—at the end of a $B_{u,v}$—conflicts with paths taking another edge, and placing code at the destination of the edge—at the beginning of a B_u—conflicts with paths arriving from another edge. When critical edges are "connected" by their source or destination basic blocks, this creates an "atomic" region (the edges) where code cannot be inserted. On this region, values cannot be moved between registers, i.e., variables must be assigned to a unique register. Hence, splitting cannot help for register allocation in these regions. We will define later in Section 3.3 these regions as *multiplexing regions* that create atomic objects hard to color. An example of such a multiplexing region is the one containing all the critical edges of Chaitin et al.'s proof: the edges from blocks of type $B_{u,v}$ to blocks of type B_u.

> *To conclude, Chaitin et al.'s original proof can be interpreted as follows. It is NP-complete to decide if the program variables can be assigned to R registers, even if live-range splitting is allowed, but only when the program has critical edges that cannot be split, i.e., when one can neither change the control-flow graph (CFG) structure nor add new basic blocks.*

3.1.3 Split points on edges

Pereira and Palsberg [2006] pointed out that the construction of Chaitin et al.—as done in Figure 3.1—is not enough to prove anything about register allocation through SSA; we will explain why in Section 3.2.1. In fact, Chaitin et al.'s proof does not hold whenever it is possible to add basic blocks on edges, and split variables using copies on these blocks. For instance, Figure 3.2 shows how to allocate the code of Figure 3.1c with 3 registers. The variable definitions of each block of type $B_{u,v}$ are arbitrarily put in 3 registers—independently of other blocks, e.g., r_1 for u, r_2 for v, and r_3 for x. Then it is decided that the variables u and x in each block of type B_u are always expected in registers r_1 and r_3. The coloring can then be "repaired" at each join point, when needed, thanks to an adequate re-mapping of registers—here a move from r_2 to r_1—in a new block along the edge from $B_{u,v}$ to B_v.

This implies that, whenever no coloration can be found for a graph, it may be possible to split some variables and some edges in order to be able to do register assignment. This has a cost, the cost of the added copies and the jumps to the new basic blocks, but which is possibly more interesting than spilling some variables to memory to make some space in registers. This lead us to the following important question for practical register allocation:

> What if both live-range splitting and critical edge splitting is allowed?

A similar question is addressed by Pereira and Palsberg [2006], to which they answer "no," the problem is still NP-complete, using a reduction from the k-colorability problem for circular-arc graphs, which is NP-complete if k is a problem input [see Garey et al., 1980, Problem GT4]. Basically, their idea is to start from a circular-arc

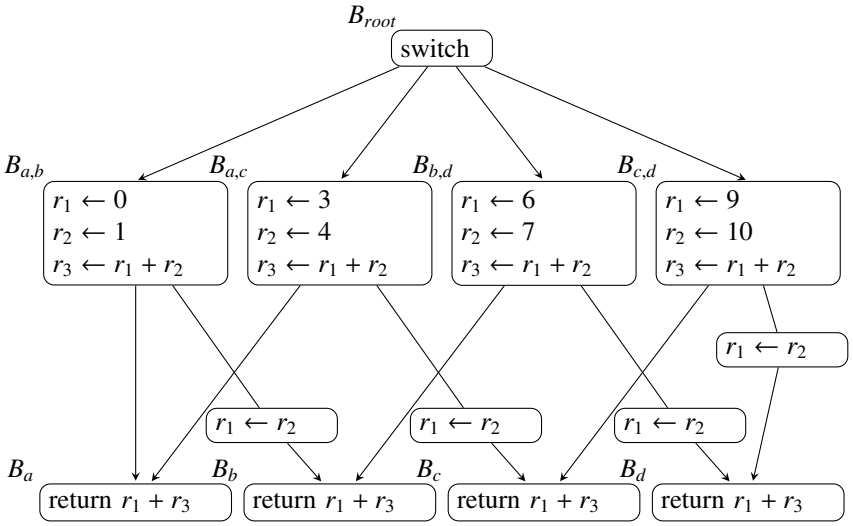


Figure 3.2: Splitting Chaitin's program makes it 3-colorable.

graph, to cut all arcs at some point to get an interval graph, to view this interval graph as the interference graph of a basic block, to add a back edge to form a loop, and to make sure that k variables are live on the back edge. This ensures that variables cannot be permuted on the back edge, supposing one needs a free register to perform permutations. Then, coloring the basic block so that no permutation is needed on the back edge is equivalent to coloring the original circular-arc graph. This is the same technique used in Garey et al. [1980] to reduce the coloring of circular-arc graphs from a permutation problem. The proof of Pereira and Palsberg shows that if we restrict to the split points defined by SSA, it is difficult to choose the right coloring of the SSA representation and thus decide if k registers are enough. It is NP-complete even for a simple loop and a single split point. However, the drawback of this proof is that, if k is fixed, this specific problem is polynomial as is the k-coloring problem of circular-arc graphs, by propagating possible permutations. We now show that, with a simple variation of Chaitin et al's proof, we can get a similar NP-completeness result, even for a fixed k, but for an arbitrary program.

Theorem 3.2. *If permutations need a free register, it is still NP-complete to decide if R register are enough, even when critical edge splitting and variable splitting at the entry or exit points of basic blocks are allowed, and even for a fixed $R \geq 3$.*

Proof. Let us consider an arbitrary graph $G = (V, E)$, and the corresponding program built using Chaitin et al.'s construction. Let us split the critical edges and add instructions to the new basic blocks as shown on Figure 3.3c. The program has three variables u, x_u, y_u for each vertex $u \in V$ and a variable $x_{u,v}$ for each edge $(u, v) \in E$. For each $(u, v) \in E$, a block $B_{u,v}$ defines u, v, and $x_{u,v}$. For each $u \in V$, a block B_u reads u, y_u, and x_u, and returns a new value. For each block $B_{u,v}$, there is a path to the blocks B_u and B_v. Along the path from $B_{u,v}$ to B_u, a block reads v and $x_{u,v}$ to define y_u, and then defines x_u. An entry block switches to all blocks $B_{u,v}$. The interference graph is now G plus some triangles: for each node v, there is a triangle consisting of (v, x_v, y_v), and for each

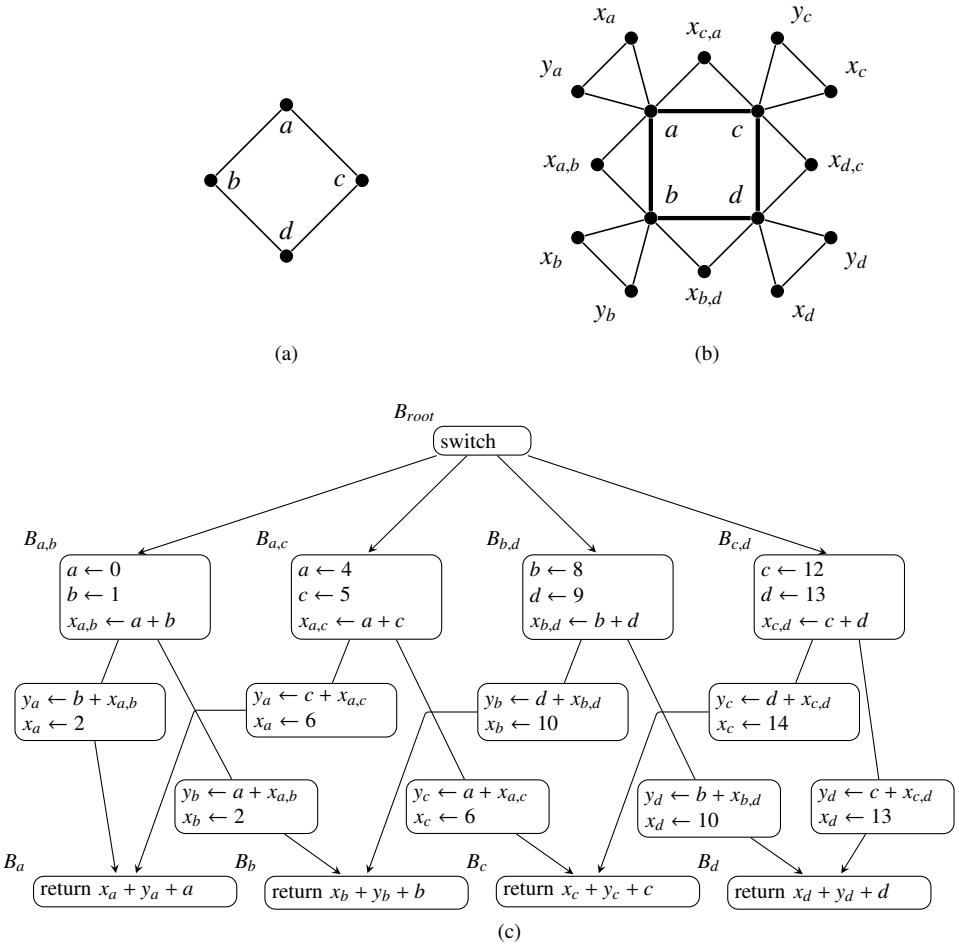


Figure 3.3: Chaitin-like construction with critical edge and variable splitting: from a cycle of length 4 (a), program (c) is built, with interference graph is (b).

edge (u, v), there is a triangle consisting of $(u, v, x_{u,v})$. See an example on Figure 3.3. Hence, the interference graph is 3-colorable iff G itself is 3-colorable.

This program does not have any critical edge, so placing permutations along the edges is equivalent to placing them on entry or exit of the intermediate blocks, between blocks of type $B_{u,v}$ and blocks of type B_u. We claim that the program can be assigned to 3 registers iff G is 3-colorable. The point is that one needs a free temporary register to perform a permutation, indeed, swapping a and b for instance requires the following instructions to be executed: $[t \leftarrow a; a \leftarrow b; b \leftarrow t]$. Since, for each u and v, exactly 3 variables are live on exit of $B_{u,v}$ and on entry of B_u and B_v, no permutation—except the identity—can be done if only 3 registers are available. Thus the live-range of any variable $u \in V$ *cannot be split*, i.e., each variable must be assigned to a unique color. Using the same color for the corresponding vertex in G gives a 3-coloring of G. Conversely, if G is 3-colorable, assign to each variable u the same color as the vertex u. It remains to color $x_{u,v}$, x_u, and y_u. This is easy: in block $B_{u,v}$, only two colors are used so far: the colors for u and v, so $x_{u,v}$ can be assigned the remaining color. Finally, x_u and y_u are assigned the two colors not used by u (see Figure 3.3b again to visualize the cliques of size 3). This gives a valid register assignment. □

To conclude, this slight variation of Chaitin et al.'s proof shows that, if the splitting of live-ranges is allowed on edges—and *only* on edges[4]—, it is still NP-complete to decide if R registers are enough. This is true even for a fixed $R \geq 3$ and even for a program without any critical edge. The proof is based on the fact that it is not possible to split at points where Live equals R.

However, we made two important assumptions in our proof: First, we allowed the splitting of variables to take place only on edges—or, equivalently, only at the entries and exits of blocks while splitting critical edges—while we forbade it inside basic blocks. This is what a traditional out-of-SSA translation does (see Section 3.2.1). Second, we assumed that one needs a free register in order to perform a swap or permutation. We argue in the next sections that these hypotheses may not be very realistic.

3.1.4 Split points anywhere

The study of Section 3.1.3 does not completely answer the question. Indeed, who said that split points need to be on entry and exit of blocks exclusively? Why not allow registers to be shuffled at any program point, for example in the middle of a basic block, if this helps performing a permutation? Consider Figure 3.3c again. The register pressure is 3 on any control-flow edge; this was the key for the proof of Section 3.1.3. But it is not 3 everywhere: it drops to 2 between the definitions of each y_u and each x_u. At this point, some register-to-register moves can be inserted to permute two colors and, thanks to this, 3 registers are always enough for such a program. One can color independently the top (including the variables y_u) and the bottom (including the variables x_v), then place adequate permutations between the definitions of y_u and x_u. This opens the way to the following question:

> Is it really NP-complete to decide if R registers are enough when splitting can be done anywhere and swaps are not available?

None of the previous proofs answers this question, and certainly not the initial proof of Chaitin et al. The problem with the previous construction is that there is no way, with

[4] Splitting at the borders of basic blocks is equivalent: it consists of splitting on every entering or leaving edge.

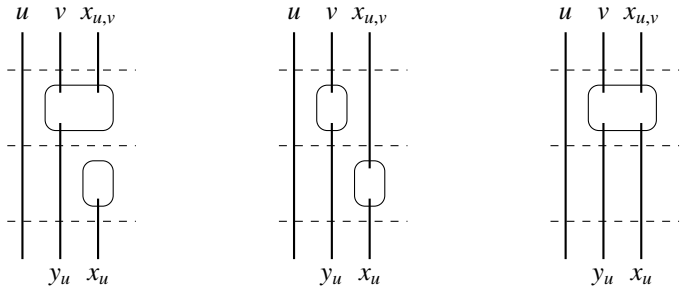


Figure 3.4: Three cases: register pressure drops to 2 (a) or is constant to 3 (b), (c).

simple statements, to avoid a program point with a low register pressure while keeping the reduction with graph 3-coloring. This is illustrated in Figure 3.4: (a) illustrates the previous situation where the register pressure drops to 2, and (b) a situation with a constant register pressure equal to 3, but that does not keep the equivalence with graph 3-coloring—$x_{u,v}$ would interfere with y_u and y_v. The only way is depicted in (c): one needs an instruction that can define more than one value. It is then easy to modify the proof and the following theorem holds.

Theorem 3.3. *If there exists instructions that can define more that one value at a time, but swaps of variables are not allowed, it is NP-complete to decide if R registers are enough, even when critical edge splitting is allowed and variable splitting is allowed anywhere.*

Proof. In the proof of Theorem 3.2, for each variable u, the variables of type x_u and y_u can be defined by a statement $(x_u, y_u) = f(v, x_{u,v})$ that consumes v and $x_{u,v}$ and produces y_u and x_u, *simultaneously*, as depicted by Figure 3.4c. Now, the register pressure is 3 everywhere in the program and thus G is 3-colorable iff the program can be mapped to 3 registers. Thus, it is NP-complete to decide if R registers are enough *if two variables can be created simultaneously* by a machine instruction and swaps are *not* available. □

In this proof, we used an instruction f capable of producing at least two values. Such a function should consume and produce the same number of values—at least 2—otherwise the register pressure would be lower just before or after it and a permutation could be inserted there. Notice the similarity with circular-arc graphs: as mentioned by Garey et al. [1980], coloring circular-arc graphs remains NP-complete even if at most 2 circular arcs start at any point, but not if only one can start.

However, it should be noticed that if such a machine instruction f exists, it is likely that a register swap is also provided in the architecture. We will discuss such architectural subtleties in Chapter 8, Section 8.1.2.1. The case where a swap instruction exists is easy since any permutation can be done. In that case, R registers are enough iff $\Omega \leq R$.

We will see later the remaining case, where register swaps are not available but *at most one* variable can be created at a given time—as it is in traditional sequential assembly-level code representation. This case does not belong to this section since it is *not* an NP-complete case.

Table 3.1: Summary of complexity proofs using Chaitin-like reductions from Section 3.1, with also polynomial results of Section 3.2.

3.1.5 Summary and discussion of complexity proofs

In the previous sections, we tried to give a pedagogic introduction to the complexity of register allocation. Starting with the original NP-completeness proof of Chaitin et al. [1981], the first step was to argue that register allocation has more freedom that graph coloring, in the sense that variables can be assigned to different registers at different points of their lives—at the cost of additional register-to-register copies. And the original proof, often cited wrongly, does not say anything regarding this view of register allocation.

Theorem 3.1 proves that even with live-range splitting, the problem remains NP-complete because code cannot be placed on critical edges. The next step was to state that code can be placed on critical edges by splitting them and adding basic blocks,[5] but Theorem 3.2 states this will not help in the absence of swap instruction since the register pressure can be increased on all edges to prevent permutations to take place. This proof did not hold if permutations could be placed anywhere and not restricted to be only on edges. But this more realistic case is not easier if it possible to define two variables at the same time, and one still does not have any swap instruction. Table 3.1 recall these results visually, along with the two polynomial results that will be described in the next section.

At this point, it seems that whatever is ingeniously included in register allocation to break the complexity piteously fails. But astute readers would have already guessed that the more freedom there is in register allocation, the more constraints there are on the architecture to keep the problem NP-complete. In particular, the last constraint is probably not very realistic (a machine capable of defines multiple values at a time, but which cannot perform a swap). We would not be offended if people started to say that, in the last theorem, we where not only splitting variables and edges, but also hairs. That is true.

But now, in the next section, we will see what can be done if we *do* have a swap instruction, or if *no* instruction can define two variables simultaneously, or when the register pressure provides a free register to perform a swap. In practice, this is nearly

[5] Discussions on whether this is possible or not will take place in Chapter 8.

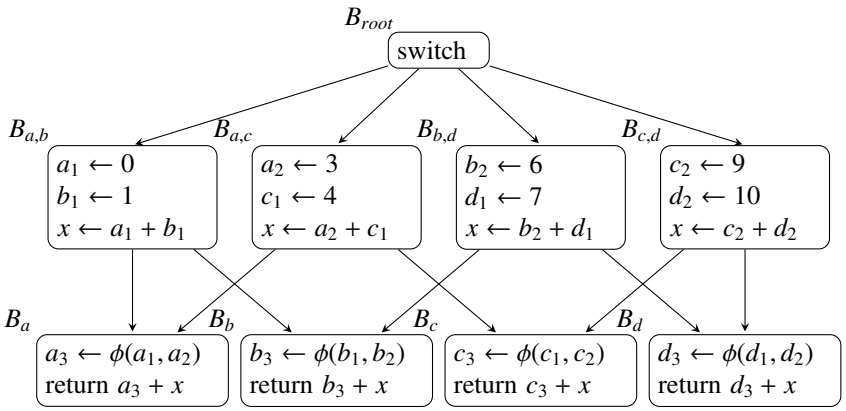


Figure 3.5: Original program of Chaitin et al.'s proof under SSA.

always the case, and under these "standard" conditions, we completely leave in fact the NP-completeness.

3.2 Polynomial solutions

In this section, we give some polynomial instances for register allocation. We also try to explain where these solutions manage to "escape" the NP-completeness of Chaitin et al.

3.2.1 Static Single Assignment

SSA was the motivation of this study, because of the recent discovery that under SSA form, the interference graph of a program is chordal, hence easy to color. We will now explain why, under SSA, we do not fall into one of the four cases of NP-completeness depicted in Table 3.1.

The easiest way to understand why is by trying to transform the program of Chaitin et al.'s original proof into SSA. This would result in the program shown on Figure 3.5, where ϕ-functions are inserted at join points of the program. The semantics of ϕ-functions, as explained in Section 2.3.6, is that copies are placed on the incoming edges. For instance, the ϕ-function $[a_3 \leftarrow \phi(a_1, a_2)]$ corresponds to adding the copy $[a_3 \leftarrow a_1]$ on the edge from $B_{a,b}$ to B_a and $[a_3 \leftarrow a_2]$ on the edge from $B_{a,c}$ to B_a. Moreover, we explained in Section 2.3.6 that the semantics of multiple ϕ-functions is that they are executed in parallel, meaning that permutation are available.

In other words, *SSA implicitly considers that critical edges can be split and that permutations can be performed on them*. Under SSA, the variable splitting only occurs on edges, but unlike the third line of Table 3.1 (Theorem 3.2), swap instructions are considered to be available. In fact, if one adds actual (parallel) copies to the program at the same place as a classical out-of-SSA conversion, the interference graph of the program becomes chordal, as is the interference graph under SSA. Because it is easy to test if k colors are sufficient to color a chordal graph (see Section 2.2.2.4), it is then easy to test if R registers are sufficient for a program under SSA. Moreover, as chordal graphs are perfect graphs, the condition is simply that Maxlive must be lower than R, i.e., $\Omega \leq R$.

3.2.2 Color propagation

Another interesting lead where to look for polynomiality comes from the fourth line of Table 3.1, i.e., Theorem 3.3. It is stated there that if swap instructions are not available, but some instructions can define multiple variables at the same time, the problem is still NP-complete. We have seen that having swap instructions makes the problem polynomial, which is what SSA assumes. We will now study the case where there is no swap instruction, but all instructions define at most one variable at a time. We believe this case is more realistic that the requirements of Theorem 3.3, but we prefer to delay these discussions until Chapter 8 in order not to lose our focus, which is trying to be as complete as possible when evaluating the possible conditions for Chaitin et al.'s proof. The next theorem states this case is polynomial.[6]

Theorem 3.4. *If blocks can be introduced to split critical edges, if live-range splitting can be done anywhere and if instructions can define at most one variable, it is polynomial to decide if R registers are enough, in the case of a strict program.*

The idea is that permutations can always be performed whenever there is a free register; and if there is none, there is no choice for coloring as explained by Figure 3.4b.

More precisely, if $\Omega > R$, it is not possible to assign the variables of a strict program to R registers without spilling, as two simultaneously live variables interfere.[7] If $\Omega < R$, it is always possible to assign variables to R registers by splitting live-ranges and performing adequate permutations. When $\Omega \leq R$, the same occurs for a point with register pressure strictly less that R: a color mismatch can always be repaired by an adequate permutation, thanks to an available register. Thus, for a strict program, the only problem may come from the sequences of program points where the register pressure remains equal to R. But, unlike Section 3.1.4 where the degree of freedom in choosing colors—at least 2—leads to NP-completeness, the fact that, here, at most *one* variable can be defined at a time simplifies the problem—the newly created variable has no choice but being assigned to the same color as the dying one the example showed previously on Figure 3.4b). This does *not* mean that R registers are always enough, but it is easy to decide if this is the case. To prove this fact precisely, we need to define formally what we mean by *color propagation*. In the following proof, we will exhibit an algorithm that answers in polynomial time the question whether R registers are sufficient or not. In should be noted that this algorithm is not intended to be used for practical register allocation: it would perform poorly as there is no mechanism to minimize the number of permutations inserted. The way colors are chosen in different connected components (randomly) would produce a lot of shuffle code between them, without any coalescing effort to remove them.

Definition 3.5 (Color propagation)**.** Liveness analysis defines, for each instruction s, $live\_in(s)$ and $live\_out(s)$, the set of variables alive just before s and just after s. These sets can be colored locally, propagating the colors from instruction to instruction, i.e., coloring variables in neighbor sets with the same color, following the control-flow forwards or backwards, i.e., considering the control-flow as undirected. More formally, coloring a statement s means defining two injective maps $col\_in(s)$

[6] Actually, it is also polynomial if instructions like $[(a,b) \leftarrow f(c)]$ exist. Indeed, only one variable is used and two are defined, which means that before f, there was one free register. This is a case for instance with a `load64` that load a 64-bits value into two 32-bits registers.

[7] Notice that it is only true for a *strict* program (we leave the non-strict case open), and with the relaxed Definition 2.8 of interference where two variables having the same value interfere.

(resp. $col\_out(s)$) from $live\_in(s)$ (resp. $live\_out(s)$) to $[1 \dots k]$. When colors propagate from a statement s_1 to a statement s_2, forwards, $col\_in(s_2)$ is defined so that $col\_in(s_2)(x) = col\_out(s_1)(x)$ for all $x \in live\_in(s_2) \cap live\_out(s_1)$ and different colors are arbitrarily picked for the other variables. The same is done to define $col\_out(s_2)$ from $col\_in(s_2)$. When propagating backwards, the situation is symmetric; $col\_out(s_2)$ is defined from $col\_in(s_1)$, then $col\_in(s_2)$ from $col\_out(s_2)$.

Below, when explaining the effect of propagation, we will assume a forward propagation; for the backward one, exchange the suffixes "in" and "out." Of course, both forward and backward propagation can appear during the execution of the algorithm.

Proof of Theorem 3.4. Let us consider only the subgraph of the control flow graph defined by the program points where the register pressure is equal to R, i.e., the propagation takes place between two instructions s_1 and s_2 such that both $live\_out(s_1)$ and $live\_in(s_2)$ have R elements. We claim that, if R registers are enough for each connected component of this graph, there is a unique solution, up to permutations of the colors, except possibly for the sets $live\_out(s_2)$ where the propagation stops ($live\_in(s_1)$ for backwards propagation). Indeed, for each connected component, start from an arbitrary program point and an arbitrary coloring of the R variables alive at this point. Propagate this coloring, as defined above, backwards and forwards along the control flow until all points of the component are reached. In this process, there is no ambiguity to choose a color: First, there is no choice for defining $col\_in(s_2)$ from $col\_out(s_1)$ since $live\_out(s_1) = live\_in(s_2)$ (in general, $live\_in(s_2) \subseteq live\_out(s_1)$ because s_1 can have more than one successor, but since both sets have R elements, they are equal); Second, if $live\_out(s_2)$ has R elements, then either $live\_out(s_2) = live\_in(s_2)$ or, as s_2 defines at most one variable, there is a unique variable in $live\_out(s_2) \setminus live\_in(s_2)$ and a unique variable in $live\_in(s_2) \setminus live\_out(s_2)$: these two variables must have the same color, and there is no choice when defining $col\_out(s_2)$ from $col\_in(s_2)$ either. Therefore, for each connected component, going backwards and forwards defines, if it exists, a *unique* solution up to the initial permutation of the colors. In other words, if there exists a solution, it can be defined by propagation for each connected component. Moreover, if propagation reaches a program point already assigned and if the colors do not match, this proves that R registers are *not* enough.

Finally, if the color propagation on each connected component provided a solution, then R registers are enough for the whole program. Indeed, the rest of the program—where register pressure is less than R—can be colored in a greedy (but not unique) fashion. Upon reaching a point already assigned, a possible color mismatch is easily repaired: an adequate permutation of colors between s_1 and s_2 is inserted: in the same basic block as s_1, if s_2 is the only successor of s_1 (resp. predecessor for backward propagation), or in the same basic block as s_2, if s_1 is the only predecessor of s_2 (resp. successor). This is always possible because there is no critical edge and there are at most $R - 1$ alive variables at this point. □

Summary of the algorithm. How to decide if R registers suffice when $\Omega \leq R$, and color when possible? First propagate colors, following the control flow along program points *where the register pressure is exactly* R . If a program point is already colored and the colors do not match, more spilling needs to be done. Otherwise, perform a second propagation phase along all remaining program points: if a program point is already colored and the colors do not match, a permutation of at most $R - 1$ registers solves the problem, using an extra available register.

3.3 Explanation of complexity

The two previous sections gave us some good insights on the conditions under which it is difficult or easy to decide if R registers are enough or not for register allocation. The last NP-completeness result, Theorem 3.3, makes us think that this problem is difficult only for very specific architectures. Such architecture should provide instructions that define more than one variable at a time, but should not allow the swapping of variables. We believe that this is not realistic at all, and discussions on this will take place in Chapter 8. On the other hand, the two polynomial instances rely on the fact that critical edges can be split so that shuffle code can be added to them. The questions are then:

"Why is it difficult when edges cannot be split?"
"Where does the complexity go when edges are split?"

A clue to answer the second question is that splitting inserts basic blocks and copies, which has a cost: the additional instructions—jumps and moves that cannot be scheduled with the rest of the code—impact the performance of the program. Trying to minimize this cost is the goal of the coalescing, which aims at removing the copies between variables in a program. Knowing if it is possible to remove all the copies is difficult since, by doing so, one would get back to the original problem again. So, the goal is to find the best trade-off, removing most of the copies while still having the benefit of the splitting, i.e, easily answering the question whether R registers are sufficient or not. By splitting, the complexity of answering this question is transferred to the register coalescing problem. Its complexity will be discussed in Chapter 5.

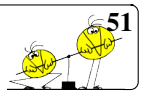
As for the first question, we already pointed out this has to do with having multiple critical edges "connected" either by their source or their destination. We will now define more clearly what we call the "multiplexing regions."

Definition 3.6. A *multiplexing region* is a maximal connected set of flow edges, where two edges are said connected iff they come from the same basic block or they go to the same basic block. Exits and entries of these basic blocks are respectively the *entry points* and *exit points* of the multiplexing region, hence defining the *borders* of this region.

It is more interesting to restrict multiplexing regions to contain only edges that cannot be split—or that one does not want to split. In that case, it can be viewed as a solid part of the program, wherein no modification can be done, instead of a collection of multiple independent program points being placed on different edges. Then, the notion of "atomic" region can be defined, which is a maximal connected set of *non-splittable* edges. From now on, we will always suppose that multiplexing regions are atomic. Otherwise, it is always possible to add empty basic blocks to edges that can be split in order to have only regions that cannot be split.

Variables of multiplexing regions must be colored, which means the interference between variables must be known on these regions, which depends directly on the notion of *liveness*. There are two kinds of variables alive on multiplexing regions:

- Variables that go through the region, i.e., which are live-in of any of the exit blocks of the multiplexing region. These variables are of course live-out of all the entry blocks that have edges going to these exits blocks since we consider only strict programs. Inside the multiplexing region, these variables are said to be alive on the edges going from the entry blocks where they are live-out to the exit blocks where they are live-in. They are called *live-through variables*.



- Under SSA, additional variables are alive on multiplexing regions: the variables defined by ϕ-functions on exit points of the multiplexing region, called *ϕ-variables*. Since the region is atomic, these variables cannot be defined on the incoming edges since that would be "inside" the multiplexing region. Hence, in the future, such a variable will have to be defined *before* entering the region and should then be considered alive on the multiplexing region, i.e., live-in of the exit block defining it, but, more importantly, also live-out of the entry blocks predecessors of the ϕ-function.[8]

It is now easy to know which variables interfere in a multiplexing region, using a definition of interference from Chapter 2. For instance, with the relaxed Definition 2.8, a and b interfere on a multiplexing region if there is a point inside the region where they are both alive, for instance on an edge, on an entry point or on an exit point. Refinements using the values of variables can be used, but this is not the point. The point is that on an atomic multiplexing region, there exists an interference graph that must be colored. A consequence of Theorem 3.1, which states that splitting variables does not help if no edge splitting is allowed, is that any graph can be the interference graph of a multiplexing region. Indeed, we will show it by re-writing more conceptually the proof of this theorem.

Proof of Theorem 3.1 using multiplexing regions. In Chaitin et al.'s proof, the critical edges going from blocks of type $B_{u,v}$ to blocks of type B_u form a multiplexing region.[9] These edges are considered non-splittable, hence the multiplexing region is atomic. Let H be the interference graph of the program (i.e., G plus $\{x\}$). Consider a splitting of the variables, and H' the corresponding interference graph. Splitting a variable can occur only outside the multiplexing region. Hence, for any variable a in H, there exists a duplicate a' of a—which can be a if it is not split in H'—such that the live-range of a' *restricted to the multiplexing region* is exactly the same as the live-range of a in the non-split program (also restricted to the region). These "micro live-ranges," the live-ranges of the duplicates restricted to the multiplexing region, are *non-splittable*, hence must be colored with a *unique* color.

In Chaitin et al.'s construction, for an edge (u, v) of G, there is a block $B_{u,v}$ with defines the two variables: they are both alive at the end of this block. This is an entry point of the multiplexing region, hence u and v interfere inside the region. So, the duplicates u' and v' of u and v alive in the multiplexing region also interfere, and $H \subseteq H'$. Hence a coloring of H' provides a coloring for H. Reciprocally, a coloring of H can be easily extended to a coloring of H' since all variables in $H' \subset H$ form independent cliques of size two or three (their live-ranges are restricted to the basic blocks). □

What are the consequences of this proof? Multiplexing regions define parts of the program that are not modifiable and their interference graph can be any graph. In atomic multiplexing regions, the variables cannot be split hence performing register allocation on these regions is NP-complete. In general, a program can be viewed as many atomic regions. Atomic regions cannot be split, and shuffle code can only be placed between them. Multiplexing regions are atomic regions, and if swaps are allowed, atomic regions inside a basic block are the instructions (no code can be added "in the middle"

[8] This view is equivalent as going out of SSA like Sreedhar et al. [1999] do, by adding copies of arguments and definition and renaming copies with a common name. The ϕ-variable would then be this common variable.

[9] That is, if G is a connected graph, which can be considered without loss of generality.

of an instruction). Else, they are contiguous program points where Live equals R.[10] Each atomic region has its "chromatic number," i.e., the minimum number of colors required to perform register assignment on the region. For a basic atomic region like an instruction, the chromatic number is simply the maximum of the size of the live-in set and the size of the live-out set. But for arbitrarily complicated multiplexing regions, it is NP-complete to compute this number.

3.4 Register allocation in two phases

This study showed two important facts. First, the difficulty comes from the presence of non-splittable critical edges. Second, if critical edges can be split, deciding if R registers are enough is easy unless under strong architectural constraints. This is not what Pereira and Palsberg [2006] proved in their article "Register allocation after classical SSA elimination is NP-complete." For their proof, they use a reduction from the k-colorability problem for circular-arc graphs, by increasing the register pressure on the back-edge of a loop so that no permutation resulting from a ϕ-function at the beginning of the loop can be performed on the back-edge. Although similar to the result of Theorem 3.1, there are two main differences with our results: First, circular-arc graph k-coloring is polynomial if k is fixed [Garey et al., 1980], while our result holds even for a fixed $k \geq 3$; Second, and more importantly, they only considered splitting at the points defined by SSA, but, as said before, we could split elsewhere.

On the contrary, our study shows that, in most cases, it is easy to decide if R registers are enough. We will see in Chapter 7 that what can be done if there are edges that cannot be split. What does this imply for register allocation? In Chapter 2, Section 2.2 was devoted to the coloring of the interference graph. We explained that, since finding if R colors are sufficient to color a graph is NP-complete, a heuristic was used: Chaitin's simplification scheme (see Section 2.2.1.2). Our study shows that, by using live-range and edge splitting, it is now possible to know in polynomial time if there is sufficiently many registers. This motivates the need for revisiting register allocation using graph coloring. Traditionally, spilling and coloring were intertwined because "how much you need to spill" was dependent on "how good you can color" the interference graph. Having a polynomial test now allows us to use algorithms with two distinct phases:

First phase. Spill variables until R registers are sufficient;

Second phase. Color variables while minimizing the number of remaining splits.

We think that separating register allocation into two independent parts gives a finer control over the problems of spilling and coalescing. We can put more effort to solve them separately, instead of having to deal with the both of them together.

A practical register allocation scheme using two phases. Critical edges can often be split. Of course this has a cost, usually the cost of an indirection—one more `jump` instruction compared to the original edge—and the cost that the code on the edge cannot be scheduled with code on other basic blocks. In that case, we present here an example of register allocation in two phases:

[10] If $\Omega > R$, we already know that R registers are not sufficient, so we consider the case where $\Omega \leq R$.

Pre-phase. Go through SSA—or any representation of live-ranges as subtrees of a tree. That is, consider that different variable definitions belong to different live-ranges.

First phase. Spill some variable if necessary. At this stage, it is easy to decide if R registers are enough: this is possible iff Maxlive[11] is less than R (because of Corollary 2.39). If R registers are not enough, additional splitting will not help as this leaves Maxlive unchanged, so spilling some variables is necessary.

Second phase. Color the variables with some coalescing to remove as many copies inserted by SSA as possible.

Post-phase. Insert the remaining copies either on new basic blocks—from split critical edges—or at the end of predecessor basic blocks for normal edges.

The first and fourth points, are called "pre-" and "post-" phases since they are not algorithmically difficult: going to SSA is a well-known exercise and adding the necessary copies is ..., well, necessary (but still requires some attention, for instance when sequentializing copies, see Section 2.3.6). *A contrario*, the phases labeled "First" and "Second" are the important and difficult ones: spilling is generally considered as a difficult problem, and the coalescing tries to minimize the number of copies that will be inserted by the post-phase.

Remaining questions. This view of coloring through permutations insertion is the base of any approach that optimizes spilling first. This approach is, for example, advocated by Knobe and Zadeck [1992]; Appel and George [2001] and Hack et al. [2006]: some spilling and splitting are done to reduce Maxlive to at most R beforehand. This approach is performed in its most extreme form by Appel and George [2001]: live-ranges are split at *every* program point in order to solve spilling optimally, hence there is a potential permutation between any two program points. But schemes in two phases like these ones—and the one we propose—leave open three questions. This thesis aims to answer these questions, at least partially:

- What (and where) to spill if R registers are not sufficient?
- How to minimize the cost of the splitting of variables and edges? *(coalescing problem)*
- What can be done if critical edges cannot be split?

3.5 Conclusion

In this chapter, we tried to clarify where the complexity of register allocation comes from. Our goal was to recall what Chaitin et al.'s original proof really proves and to extend this result. The main question addressed by Chaitin et al. is of the following type:

> Can we decide if R registers are enough for a given program or if some spilling is necessary?

[11] Using Definition 2.8 of interference, where values are not taken into account. One can assume a copy folding pass was done under SSA to rename equal variables with a common name.

3.5.1 Summary of Results

The original proof of Chaitin et al. [1981] proves that the register allocation problem is NP-complete when live-range splitting is not allowed, i.e., if each variable can be assigned to only one register. We showed that the same construction proves more: the problem remains NP-complete when live-range splitting is allowed but not (critical) edge splitting.

Recently, Pereira and Palsberg [2006] proved that, if the program is a simple loop, the problem is NP-complete if live-range splitting is allowed but only on a block on the back edge, and only if register swaps are not available. This is a particular form of register allocation through SSA. The problem is NP-complete if R is a problem input. We showed that Chaitin et al.'s proof can be extended to show a bit more. When register swaps are not available, the problem is NP-complete for a fixed $R \geq 3$ (but for a general CFG), even if the program has no critical edge and if live-range splitting can be done on any control-flow edge, i.e., on entry and exit of blocks, but not inside basic blocks.

These results do not address the general case where live-range splitting can be done anywhere, including *inside* basic blocks. We showed that the problem remains NP-complete only if some instructions can define two variables at the same time but register swaps are not available. Such a situation might not be so common in practice. For a strict program, we can answer the remaining cases in polynomial time. If Maxlive $= R$ and register swaps are available, or if Maxlive $< R$, then R registers are enough. If register swaps are not available and at most one variable can be defined at a given program point, then a simple greedy approach can be used to decide if R registers are enough.

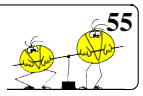
This study shows that the NP-completeness of register allocation is *not* due to the coloring phase, as may suggest a misinterpretation of the reduction of Chaitin et al. from Graph k-Coloring. If live-range splitting is taken into account, deciding if R registers are enough or if some spilling is necessary is not as hard as one might think. The NP-completeness of register allocation is due to three factors: the presence of critical edges which create multiplexing regions where variables are hard to color if they are non-splittable, the optimization of spill costs (if R registers are not enough) and of coalescing costs, i.e., choosing which live-ranges should be merged while keeping the graph R-colorable.

3.5.2 Organization of the thesis

In this thesis, we defend the idea of performing register allocation in two phases—first spilling then coloring using coalescing—instead of the classical scheme that intermixes everything in a unique phase. While the classical scheme has the advantage of being very simple in its original form (by Chaitin [1982]), or in the improved Iterated Register Coalescing (IRC) version by George and Appel [1996], it was designed this way mainly because:

- Spilling depends on whether the coloring heuristic will work or not.
- Coalescing can help Chaitin et al.'s coloring heuristic.

But one disadvantage is that changes in the scheme are difficult to implement as the whole allocator needs to be compliant: phases must be iterated (spilling introduces new variables at stores and loads), coloring depends on spilling which depends on coloring, etc.



The discovery that the interference graph of a program under SSA is chordal opened new doors for the study of splitting techniques that simplify the coloring test: "Are R registers sufficient for allocation?" With such techniques, spilling does not depend on the result of a coloring heuristic anymore: we know exactly *when* it is required or not, which breaks the first reason why register allocation is classically performed in only one complex phase. Advantages for register allocation in two phases are multiple: better control over each of the phases, no interplay between these phases—hence an easier implementation since improvements on one phase are easier to try and to evaluate.

The rest of the thesis will be organized as follows. As the spill problem is difficult for a general program, we will study its complexity for SSA programs in Chapter 4. We have indeed seen that SSA is a useful splitting technique, and we figured it would be pertinent to know better the complexity of the spill problem for programs under SSA form. Then we will study the complexity of the coalescing problem in Chapter 5, which is the important optimization of the second phase in register allocation in two phases. In Chapter 6, we will present advanced techniques for coalescing. Finally, Chapter 7 discusses the problem of non-splittable edges and permutation motion—a technique to move added copies away from critical edges. In Chapter 8, the conclusion, we will discuss practical subtleties for "real-world" register allocation in two phases.

Fear to let fall a drop and you will spill a lot.

Malayan proverb

Des petits trous, des petits trous, toujours des petits trous...

Serge Gainsbourg

4

On the complexity of spill everywhere under SSA Form

Preliminary note: this chapter is very technical but we felt it was more logical to place it there, before the coalescing chapters, so as to keep the same order as in register allocation in two phases: first, spilling, then, coalescing. This chapter is a purely theoretical study on the complexity of the spilling problem. We will not propose any practical solution, while we do so for the coalescing problem. As a consequence, it is possible to skip it at first reading.

The dominance property of Static Single Assignment (SSA) form suggests promising directions for the design of new register allocation heuristics, in particular, it is possible to cleanly separate register allocation in two phases. This was already mentioned in Chapter 2, and the study of Chaitin et al.'s NP-completeness proof in Chapter 3 explained that the SSA form simplifies the problem of knowing whether R registers are sufficient or not for the register allocation problem because it splits variables (explicitly) and edges (implicitly with ϕ-functions). However, the problem of what to do when there is not enough registers is not answered. We *do* know that, in that case, spilling some variables to memory is necessary, but not yet *how* to do it.

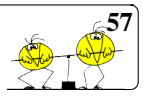
In this chapter, we will study the spill problem for programs under SSA form. The motivation of this study is driven by the hope of designing both fast and efficient register allocation in two phases—first spilling, then coloring—based on SSA form. As explained in the previous chapters, under SSA form, the test that tells whether some spilling is required or not is simply that Maxlive must be at most the number of registers: $\Omega \leq R$ (see Corollary 2.39).[1] Answering whether spilling is necessary or not is *easy* while minimizing the amount of `load` and `store` instructions is the real issue. In other words, if the search space is now cleanly delimited, the objective function that corresponds to minimizing the spill cost has still some open issues. The question is:

"Is the spilling problem easier to solve under SSA?"

The spilling problem can be considered at different granularity levels: at the highest, the so-called *spill everywhere* considers the live-range of a variable as an atomic object, i.e., a variable is either entirely spilled or entirely not spilled. This simplification consists in answering the question "what to spill," but not "where to spill." This is the same approximation made by Chaitin et al. [1981] in their NP-completeness proof and coloring algorithm. With spill everywhere, a spilled variable will stay so on its entire life, but for the `store` after the definition and the `load` before each use.[2] The

[1] Unless for very particular cases, for instance if there is no swap instruction. See Chapter 3 for details.

[2] Although, in Chaitin et al.'s algorithm, they have a mechanism to avoid reloading variables *a posteriori*, i.e., after the spill everywhere decision.



finer granularity, known as load-store optimization, optimizes each `load` and `store` separately, and in particular the placement of these instructions. The latter problem is also known as "paging with write back" and was proven NP-complete by Farach-Colton and Liberatore [2000] for a basic block, even under SSA form, when the number R of registers is an input of the problem. The former problem is much simpler, and a well-known polynomial instance by Belady [1966] exists under SSA form on a basic block. To develop new spilling heuristics, studying the complexity of spilling everywhere is very important for the design of either aggressive or just-in-time (JIT) register allocators because of the two following reasons:

1. First, the complexity of the load-store optimization problem comes from the asymmetry between `load`s and `store`s [Farach-Colton and Liberatore, 2000]. The main difference between the load-store optimization problem and the spill everywhere problem comes from this asymmetry. We measured in practice that most SSA variables have only one or two uses, so it is natural to wonder whether this singularity makes the load-store optimization problem simpler or not: for instance, in the most extreme case, with only one use per variable, this problem is equivalent to the spill everywhere problem.[3] More generally, even in the context of a traditional compiler, the spill everywhere problem can be seen as an oracle for the load-store optimization problem to answer whether a variable should be stored or not. Then, one could imagine a pass that tries to optimize the placement of loads and stores for variables chosen to be spilled. In the context of aggressive compilation using integer linear programming (ILP), [David W. Goodwin and Kent D. Wilken, 1996; Fu and Wilken, 2002; Barik et al., 2007], a way to decrease the complexity is to restore the symmetry between loads and stores as done by Appel and George [2001].[4]

2. Second, we think that the spill everywhere is a good candidate for designing simple and fast heuristics for JIT compilation on embedded systems. Again, in this context, the complexity and the footprint of the compiler is an issue. Spilling only parts of the live-ranges, as opposed to spilling everywhere, leads to irregular live-range splitting and the insertion of shuffle code to repair inconsistencies, in addition to maintaining liveness information for coalescing purpose. All of this is probably too costly for some embedded compilers.

To our knowledge, this is the first exhaustive study of the *complexity* of the spill everywhere problem in the context of SSA form in the literature.

The rest of the chapter is organized as follows. For our study, we consider different variants of the spilling problem, Section 4.1 provides the terminology and notation that describe the different cases we considered. Section 4.2 considers the simplified spill model where a spilled variable frees a register for its whole live-range; we provide an exhaustive study of its complexity under SSA form. Section 4.3 deals with the problem where a spilled variable might still need to reside in a register at its points of definition and uses; the study is there restricted to basic blocks as it is already NP-complete for this simple case. Section 4.4 summarizes our results and concludes the chapter.

[3] Supposing the frequencies of execution of basic blocks are the same.

[4] In their formulation, a variable might be either in a memory location or in a register, but cannot reside in both.

4.1 Terminology and Notation

In our study, we (almost) only consider the "everywhere" approximation of the spill problem. In this approach, the goal is to decrease the register pressure below the number of register at every program point, while minimizing the cost of the spilling, i.e., the sum of the weights of the spilled variables. For the purpose of our study, we consider three different varying parameters.

Global vs. local: Live-ranges can be *local* (i.e., only on basic blocks) or *global*. On a basic block, the interference graph is an interval graph, while it is chordal for a general control-flow graph (CFG) under SSA form with dominance property.

Memory instructions vs. store/reload: The use of an evicted (spilled) variable in an instruction may requires a register (RISC-like architecture) or not (CISC-like architecture). If it does not, spilling a variable decreases by one the register pressure on every point of the corresponding live-range. Otherwise, spilling a variable decreases the register pressure only on program points that do *not* use or define it. In the first case, spilling a variable has the effect of removing the entire live-range; in the second case, it has the effect of removing a version of the live-range with "holes" at the use and definition points (see Section 4.3). We denote these two problems respectively as spilling *without holes* or spilling *with holes*.

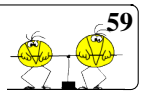
Weighted vs. unweighted: Finally, $w(v)$ denote the weight of variable v, i.e., the cost of spilling v. We distinguish the cases where the cost of spilling is the same for all variables or not. We denote these two problems respectively as *unweighted*, denoted by $w = 1$ (meaning $w(v) = 1$ for all v), or *weighted*, denoted by $w \neq 1$.

In this study, we play with these parameters, trying to make the problem more complex to see if a polynomial algorithm can still apply, or the converse, trying to simplify an NP-complete problem to see if it stays so. We always tried to give the proof that is the most constraining for the result. For instance, if a problem is NP-complete in the two cases $w = 1$ and $w \neq 1$, the proof will consider the unweighted case. Conversely, if a problem is polynomial in the two cases $w = 1$ and $w \neq 1$, the algorithm will explain how to deal with the weighted case. Remember that stronger results imply the weaker ones. This is the reason why, on tables summarizing the complexity results (namely, Tables 4.1 and 4.2, which will be introduced later), many cells are empty but nevertheless stated as "polynomial" or "NP-complete." Their status is subsumed by a stronger result, which is the one given in the colored area.

As mentioned earlier, the goal of the spilling problem is simply the problem of lowering the register pressure so that register allocation gets feasible. Under SSA form, it is necessary and sufficient to lower the register pressure so that, at every program point, it becomes less that the number of registers R. The corresponding optimization problem is to *minimize the spilling cost*. Maxlive, the maximum over all program points, will be denoted by Ω. Hence formally, the goal is to decrease Ω by spilling some variables. If we denote by Ω' the register pressure after this spilling phase, we distinguished four different problems.

Decreasing Maxlive: spill so that:

- $\Omega' \leq \Omega - 1$: "incremental spilling;"
- $\Omega' \leq \Omega - C$ where C is a constant: spill with "many registers;"



- $\Omega' \leq C$ where C is a constant: spill with "few registers;"
- and the general problem, $\Omega' \leq R$ where there is no constraint on the number of registers R.

A graph problem: The spill everywhere problem without holes can be expressed as a node deletion problem [Yannakakis, 1978]. The general node deletion problem can be stated as follows: "Given a graph or digraph G, find a set of nodes of minimum cardinal whose deletion results in a subgraph or subdigraph satisfying the property π." Hence, the results of the first section have a domain of application not only on register allocation but also on graph theory. For this reason, we formalize the results using graphs (i.e., properties of the interference graphs) instead of programs (i.e., register pressure on the CFG) while the algorithmic behind is actually based on the CFG representation.

4.2 Spill Everywhere without Holes

On a basic block, the unweighted spill everywhere problem without holes is polynomial: this is the greedy "furthest use" algorithm described by Belady [1966]. It is less known that the weighted version, which cannot be solved using this last technique, is also polynomial [Yannakakis and Gavril, 1987; Farach-Colton and Liberatore, 2000]. The interference graph is an intersection graph for which the incidence matrix is totally unimodular and the ILP formulation can be solved in polynomial time, for example using flow algorithms. This property holds also for a path graph, which is a class of intersection graphs between interval graphs and chordal graphs. We recall these results here for completeness. We also recalled earlier that, under SSA form, once the register pressure has been lowered to R at every program point, the coloring "everywhere" problem (each variable is assigned to a *unique* register) is polynomial.

The natural question raised by these remarks is whether the spill everywhere problem without holes is polynomial or not under SSA form. In other words, does the SSA form make this problem simpler? The answer is "no." A graph theory result of Yannakakis and Gavril [1987] shows it is NP-complete, even in its unweighted version: for an arbitrarily large number of registers R, a program with Ω arbitrarily larger than R, spilling everywhere a minimum number of variables such that Ω' (i.e., Ω after spilling) is at most R is NP-complete. The main result of this section shows more: this problem remains NP-complete even if one only requires Maxlive to be lowered by one, i.e., $\Omega' \leq \Omega - 1$. The practical implication of this result is that for a heuristic that would lower Ω one by one iteratively, even the optimization of each separate step is an NP-complete problem.[5]

4.2.1 Complexity results

Table 4.1 summarizes the complexity results of spilling everywhere (without holes). We will now recall classical results and prove new results, more accurate. Let us start with the decision problem related to the most general case of spill everywhere without holes.

[5] Note that providing an optimal solution for each intermediate step (going from Ω to $\Omega - 1$, then from $\Omega - 1$ to $\Omega - 2$, and so on, until $\Omega' = R$) does not always give an optimal solution for the problem of going from Ω to R.

	weighted	$\Omega' \leq C$	$\Omega' \leq R$	$\Omega' \leq \Omega - 1$
Chordal graph = *general SSA case*	no		(4.3)	X3C 4.6
	yes	dyn. prog. 4.4		
Interval graph = *basic block*	no		furthest use 4.1	
	yes		ILP 4.2	dyn. prog. 4.5

polynomial NP-complete new result

Table 4.1: Spill everywhere without holes. References to theorems are given in gray. All cases of a colored area are subsumed by the proof given in this area.

Problem: SPILL EVERYWHERE FOR PERFECT GRAPHS
Instance. A perfect graph $G = (V, E)$ with clique number $\Omega = \omega(G)$, a weight function $w(v) > 0$ for each vertex v, an integer R, an integer K.
Question. Is there a set of vertices $V_S \subseteq V$ with overall weight $\sum_{v \in V_S} w(v) \leq K$ such that the clique number Ω' of the induced subgraph G' with vertices $V \setminus V_S$ is at most R?

Theorem 4.1 (Furthest First)**.** *The spill everywhere problem for an interval graph is polynomially solvable with a greedy algorithm if $w(v) = 1$ for all v even if R is not fixed (i.e., is an input of the problem).*

The algorithm behind this theorem is the well-known "furthest use" strategy described by Belady [1966], since interference graph of variables on a basic block is an interval graph. This strategy is very interesting for designing spilling heuristics on the dominance tree (see for example Hack et al. [2005]). We give here a constructive proof for completeness.

Proof. An interval graph is the intersection graph of a family of intervals (on a straight line). For convenience, we denote by B (for "basic block") the union of all intervals. The set of intervals is denoted by V (for "variables"). B is composed of m successive "points." $p_1, \ldots, p_m$, so that intervals start and end between successive points. Once variables are removed (spilled), the set of remaining variables is called V'. The goal is to remove the minimum number of intervals so that for each point p of B, the number of intervals in V' intersecting p is at most R.

The greedy algorithm can be described as follows:

Step 0 (init) Let $V'_0 = V$ and $i = 1$;

Step 1 (find first) Let $p(i)$ be the first point from the beginning of B such that more than R variables of V'_{i-1} intersect $p(i)$; Stop if there is no such $p(i)$;

Step 2 (remove furthest) Select a variable v_i that intersects p and ends the furthest and remove it: $V'_i = V'_{i-1} \setminus \{v_i\}$;

Step 3 (iterate) Increment i by 1 and go to Step 1.

Let us prove that the solution obtained by the greedy algorithm is optimal. Consider an optimal solution S (described by a set V_S of spilled variables) such that V_S contains

the maximum number of variables v_i selected by the greedy algorithm. Suppose that S does not spill all of them and denote by v_{i_0} the variable with smallest index such that $v_{i_0} \notin V_S$. By definition of p_{i_0} in the greedy algorithm, there are at least $R+1$ variables not in $\{v_1, \ldots, v_{i_0-1}\}$ intersecting $p(i_0)$. As S is a solution, there is a variable v in V_S (thus $v \neq v_{i_0}$) that intersects $p(i_0)$. We claim that spilling $W = (V_S \setminus \{v\}) \cup \{v_{i_0}\}$, i.e., spilling v_{i_0} instead of v, is an optimal solution too. Indeed, for all points before $p(i_0)$ (excluded), the number of variables in $V'_{i_0-1} = V \setminus \{v_1, \ldots, v_{i_0-1}\}$ is at most R. Since $\{v_1, \ldots, v_{i_0}\} \subseteq W$, this is true for $V \setminus W$ too. Furthermore, each point p after $p(i_0)$ (included) that is intersected by v is also intersected by v_{i_0} by definition of v_{i_0}. Since at most R variables of $V \setminus V_S$ intersect any such p, the same is true for $V \setminus W$. Finally, this optimal solution spills more variables v_i selected by the greedy algorithm than S, which is not possible by definition of S. Thus V_S contains all variables v_i and, by optimality, only these. This proves that the greedy algorithm gives an optimal solution. □

Theorem 4.2 (poly. ILP)**.** *The spill everywhere problem for an interval graph is polynomially solvable even if $w \neq 1$ and R is not fixed.*

This result was pointed out by Yannakakis and Gavril [1987] and used in a slightly different context by Farach-Colton and Liberatore [2000]. The idea is to formulate the problem using ILP and to remark that the matrix defining the constraints is totally unimodular. For the sake of completeness, we provide the formulation here.[6]

Proof. We use the same notations as for Theorem 4.1 except that, now, $v_1, \ldots, v_n$ denote all variables and not only those selected by the greedy algorithm. Let w_i be the cost of removing (spilling) variable v_i. We define the clique matrix as the matrix $\mathcal{M} = \left(c_{p,v}\right)$ where $c_{p,v} = 1$ if v intersects the point p and $c_{p,v} = 0$ otherwise. Such a matrix is called the incidence matrix of the interval hyper-graph and is totally unimodular [Berge, 1973]. In our case, $\mathcal{M}$ is of polynomial size. This is not the case for all graphs since the number of maximal cliques, hence the number of lines of M, can be exponential, but this is not the case for interval and chordal graphs. The optimization problem can be solved using the following integer linear program, where $\vec{x}$ is a vector with components $(x_i)_{1 \leq i \leq n}$, $\vec{w}$ is a vector with components $(w_i)_{1 \leq i \leq n}$, $\vec{R}$ is a vector whose components are all equal to R, and vector inequalities are to be understood component-wise:

$$\max \left\{ \vec{w}.\vec{x} \mid \mathcal{M} \cdot \vec{x} \leq \vec{R},\ \vec{0} \leq \vec{x} \leq \vec{1} \right\}$$

Of course, $x_i = 0$ means that v_i should be removed while $x_i = 1$ means it should be kept. The matrix of the system is $\mathcal{M}$ with some additional identity matrices, which keeps the total unimodularity. □

The next theorem is from Yannakakis and Gavril [1987]. While their formulation of the problem is different—they search for k-colorable subgraphs in chordal graphs—, they deal in fact with the same problem as us. We refer to their paper for the proof as we will improve this result in Theorem 4.6.

Theorem 4.3 (Yannakakis)**.** *The spill everywhere problem for a chordal graph is NP-complete even if $w(v) = 1$ for each $v \in V$.*

Another important result of Yannakakis and Gavril [1987] is that the spill everywhere problem is polynomially solvable when R is fixed. Of course, there is a power of R in the complexity of their algorithm, but it means that if R is small, the problem is

[6] Note that Farach-Colton and Liberatore [2000] also have a flow formulation for this problem.

simpler. Because of this, we call the problem when R is fixed "spill everywhere *with few registers*".

Problem: SPILL EVERYWHERE WITH FEW REGISTERS (C)
Instance. A perfect graph $G = (V, E)$ with clique number Ω, a weight $w(v) > 0$ for each vertex, an integer K, $R = C$ is fixed (i.e., a constant, not an input of the problem).
Question. Is there a subset V_S of vertices V with overall weight $\sum_{v \in V_s} w(v) \leq K$ such that the clique number of the subgraph G' induced by $V \setminus V_S$ is $\Omega' \leq R$?

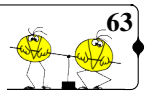
Theorem 4.4 (Dynamic programming on non-spilled variables)**.** *The spill everywhere problem with few registers ($R = C$) is polynomially solvable if G is chordal even if $w \neq 1$.*

When we proved our results, we were actually not aware of Gavril and Yannakakis paper. Since Theorem 4.4 is very intuitive, we logically ended with the same kind of construction. For completeness, we provide it here, with our own notations. This proof is constructive and the algorithm (dynamic programming on program points) is based on a tree traversal. The idea is that at each point, the number of variables *not spilled* is at most C, hence there are at most Ω^C possibilities at every program point. Of course, this works only because the underlying structure is a tree: solutions of children are independent and can be "glued" together during the dynamic programming. This could not be done if there was cycles of program points. The algorithm performs $O\left(m\Omega^C\right)$ steps of dynamic programming, where m is the number of program points.

Proof. A chordal graph is the intersection graph of a family V of subtrees of a tree T [Golumbic, 1980, Thm 4.8]. We call *points* the vertices of the tree T, and for each point p, T_p the maximal subtree of T rooted at p (the root is r, and $T_r = T$). To distinguish the subtrees T_p from the subtrees of the family V, we call the latter *variables*. Given a point p and a subset $W \subseteq V$ of variables, let $W(p)$ be the set of variables $v \in W$ intersecting p, i.e., such that p belongs to the subtree v. If $|W(p)| \leq C$, we say that W fits p or that $W(p)$ is a fitting set for p. We say that W fits a set of points if it fits each of these points. A solution to the spill everywhere problem with C registers is thus a subset W of V such that W fits T. It is an optimal solution if $\sum_{v \in W} w(v)$ is maximal. With these notations, W corresponds to $V \setminus V_S$ in the spill everywhere problem formulation, and maximizing the cost of W is equivalent to minimizing the weight of V_S.

Given a subset of variables W, we consider its *restriction*, denoted by W_p, to a subtree T_p: it is defined as the set of variables $v \in W$ that have a non-empty intersection with T_p. Note that if W fits T, then its restriction W_p to a subtree T_p fits T_p. Furthermore, if p_1 and p_2 are children of p in T then, because of the tree structure, all variables that belong to both W_{p_1} and W_{p_2} intersect p. Also, for $i \in \{1, 2\}$, all variables in W_{p_i} intersecting p intersect also p_i, i.e., $W_{p_i}(p) = W_p(p_i)$. These remarks ensure the following. Let W be a fitting set for T_p and let W' be a fitting set for T_{p_i} such that $W'_{p_i}(p) = W_{p_i}(p)$ (i.e., they coincide between p and p_i). Then, replacing W_{p_i} by W'_{p_i} in W leads to another fitting set of T_p. This is the key to get an optimal solution thanks to dynamic programming.

The final proof is an induction on the points p of T—from the leaves to the root—and on the fitting sets $F_p \in \mathcal{F}_p = \{W \subseteq V(p); |W| \leq C\}$ of these points. Let us denote by $W_{max}(p, F_p)$ a subset W of V that contains only variables intersecting T_p, such that $W(p) = F_p$, and with maximal cost. The goal is to calculate, for all points p and all fitting set $F_p \in \mathcal{F}_p$, $W_{max}(p, F_p)$. Then, the cost of the best solution for the whole tree



will be the maximum of $W_{max}(r, F_r)$, which defines a fitting set $F_{r_{max}}$. The best spill solution is then easily found by a tree traversal starting at r with solution $F_{r_{max}}$, going down to the leaves, using the previous dynamic programming computations.

The hard part is to compute, for a given point p and one of its fitting sets $F_p \in \mathcal{F}_p$, $W_{max}(p, F_p)$. If p is a leaf, it is easy since there is no much choice but counting the number of variables in F_p. If p is not a leaf, it has at least one child. The solutions for each child of p are independent but for the variables they have in common. Since this is a tree, these variables intersect p, hence their status (spilled or non-spilled) is governed by the fitting set F_p. Then, for each child p_i of p, we only need to consider the fitting sets F_{p_i} that match F_p, i.e., such that $F_{p_i} \cap V(p) = F_p \cap V(p_i)$. From the remark above (with W and W'), any of such sets can be plugged on F_p, and this can be done independently for each child. Now, we need to find, for all combinations of fitting sets F_{p_i} for each children, which one gives the greatest cost. For a combination, the cost is easy to compute: it is the cost of F_p plus, for each child, the cost of W_{p_i} minus the variables in common between F_p and F_{p_i}, i.e.:

$$\begin{aligned}
\text{cost of } W_p &= \sum_{v \in F_p} w(v) + \sum_{p_i} \left(\sum_{v \in W_{p_i} \setminus F_p} w(v) \right) \\
&= \sum_{v \in F_p} w(v) + \sum_{p_i} \left(\text{cost of } W_{p_i} - \sum_{v \in F_{p_i} \cap F_p} w(v) \right)
\end{aligned}$$

One should note that, in the cost of W_p, the cost of the solution brought by one child in independent from the solutions brought by other children. Hence, we can optimize independently the solution of each child. For a child p_i, and for a set F_{p_i} that matches F_p, the cost of the variables in common between F_p and F_{p_i} (the part after the minus sign in the equation above) is a constant, hence it is best to maximize the part "cost of W_{p_i}." For that, we just need know $W_{max}(p_i, F_{p_i})$, which is ensured by the dynamic programming. So, we pick the F_{p_i} such that $W_{max}(p_i, F_{p_i})$ is maximal. From these selected subsets, one for each p_i, we construct $W_{max}(p, F_p)$.

This construction is done for each $F_p \in \mathcal{F}_p$. As there are at most $V(p)^C \leq \Omega^C$ such fitting sets for p, these successive locally optimal solutions can be built in polynomial time. □

We have just seen that, whenever the number of register is fixed, the spill problem can be solved polynomially. However, the complexity grows exponentially with R so it works best with a very small number of registers. This might be a clue to explain why aggressive techniques like ILP for spilling appeared to work during the last decade for instance for X86. Appel and George [2001] were probably well aware of this fact when they entitled their article "Optimal spilling for CISC machines *with few registers*." We now address the following problem, which is a particular case of the more general spill everywhere problem.

Problem: INCREMENTAL SPILL EVERYWHERE
Instance. A perfect graph $G = (V, E)$ with clique number $\Omega = \omega(G)$, a weight $w(v) > 0$ for each vertex, an integer K.
Question. Is it possible to remove vertices $V_S \subseteq V$ from G with overall weight $\sum_{v \in V_s} w(v) \leq K$ such that the induced subgraph G' has clique number $\Omega' \leq \Omega - 1$?

The following theorem can be seen as a particular case of Theorem 4.2. The proof is interesting since it provides an alternative solution to the ILP formulation for this simpler case.

Theorem 4.5 (Dynamic programming on spilled variables)**.** *If G is an interval graph, the incremental spill everywhere problem is polynomially solvable, even if $w \neq 1$.*

Proof. Let $B = \{p_1, \ldots, p_m\}$ be a linear sequence of points, $p_i < p_j$ if $i < j$, and $V = \{v_1, \ldots, v_n\}$ be a set of weighted variables, where each variable v_i corresponds to an interval $[s(v_i), e(v_i)]$. We assume that the variables are sorted by increasing starts, i.e., $s(v_i) \leq s(v_j)$ if $i < j$. Without loss of generality, the problem can be restricted to the case where any point p belongs to exactly Ω variables (any other point can be deleted from the instance). So for each point, one needs to spill at least one of the intersecting variables. What we seek is thus a minimum weighted cover of B by the variables of V, which can be done thanks to dynamic programming as follows.

Let $W(p_i)$ be the minimum cost of a cover of $p_1, \ldots, p_i$. Knowing all $W(p_{j<i})$, it is possible to compute $W(p_i)$. Indeed, at p_i, one must choose a variable $v \in V(p_i)$, i.e., intersecting the point p_i. As v already covers the interval between its start $s(v)$ and p_i, we get:

$$W(p_i) = \min_{v \in V(p_i)} (w(v) + W(\text{pred}[s(v)])) \text{ where } \text{pred}[p_i] = p_{i-1}$$

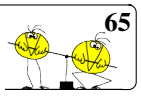
with the convention $W(p) = 0$ for $p < p_1$. $W(p_m)$ is the minimum cost of an incremental spilling over the whole basic block B. The set $V(p_i)$ can be computed from $V(p_{i-1})$ in $O(\Omega)$ operations because the variables are sorted by increasing starts. The overall complexity is thus $O(\Omega m)$. □

We will now show you a stronger theorem than the Theorem 4.3 of Yannakakis and Gavril [1987]. We discovered it while following our first (false) intuition, which was that choosing which variables to remove so as to go from Ω to $\Omega - C$ was exactly the symmetric of choosing which variables to keep so as to get down to C (with C being a constant). At first sight, it seemed that dynamic programming could be used, as for Theorem 4.4, to solve the incremental spill everywhere problem. For interval graphs, both problems can indeed be solved with dynamic programming as we previously showed. The incremental approach would have then provided a heuristic for the main spill everywhere problem, as an alternative to an exact solution as in Appel and George [2001], which is too expensive when R is large. Unfortunately, Theorem 4.6 contradicts this intuition.

Theorem 4.6 (From 3-exact cover (X3C))**.** *The incremental spill everywhere problem is NP-complete for a chordal graph even if $w(v) = 1$ for each $v \in V$.*

Proof. As for Theorem 4.4 we use the characterization of a chordal graph as an intersection graph of a family of subtrees of a tree. We use the same notations. The proof is a reduction from Exact Cover by 3-Sets (X3C) [Garey and Johnson, 1979, Problem SP2]: let $\mathcal{P}$ be a set of $3n$ elements $\{p_1, p_2, \ldots, p_{3n}\}$, and $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ a set of subsets of $\mathcal{P}$ where each subset contains exactly three elements of $\mathcal{P}$. Does $\mathcal{V}$ contains an exact cover of $\mathcal{P}$, i.e., a sub-collection $\mathcal{S} \subseteq \mathcal{V}$ such that every element of $\mathcal{P}$ occurs in exactly one member of $\mathcal{S}$?

Let us consider an instance of X3C and define the following family of subtrees of a tree (see Figure 4.1): the main tree T is of height 2 with one root point labeled p_0 and $3n$ leaves labeled $p_1, p_2, \ldots, p_{3n}$. For each $v_i = \{p_\alpha, p_\beta, p_\gamma\}$ there is a subtree



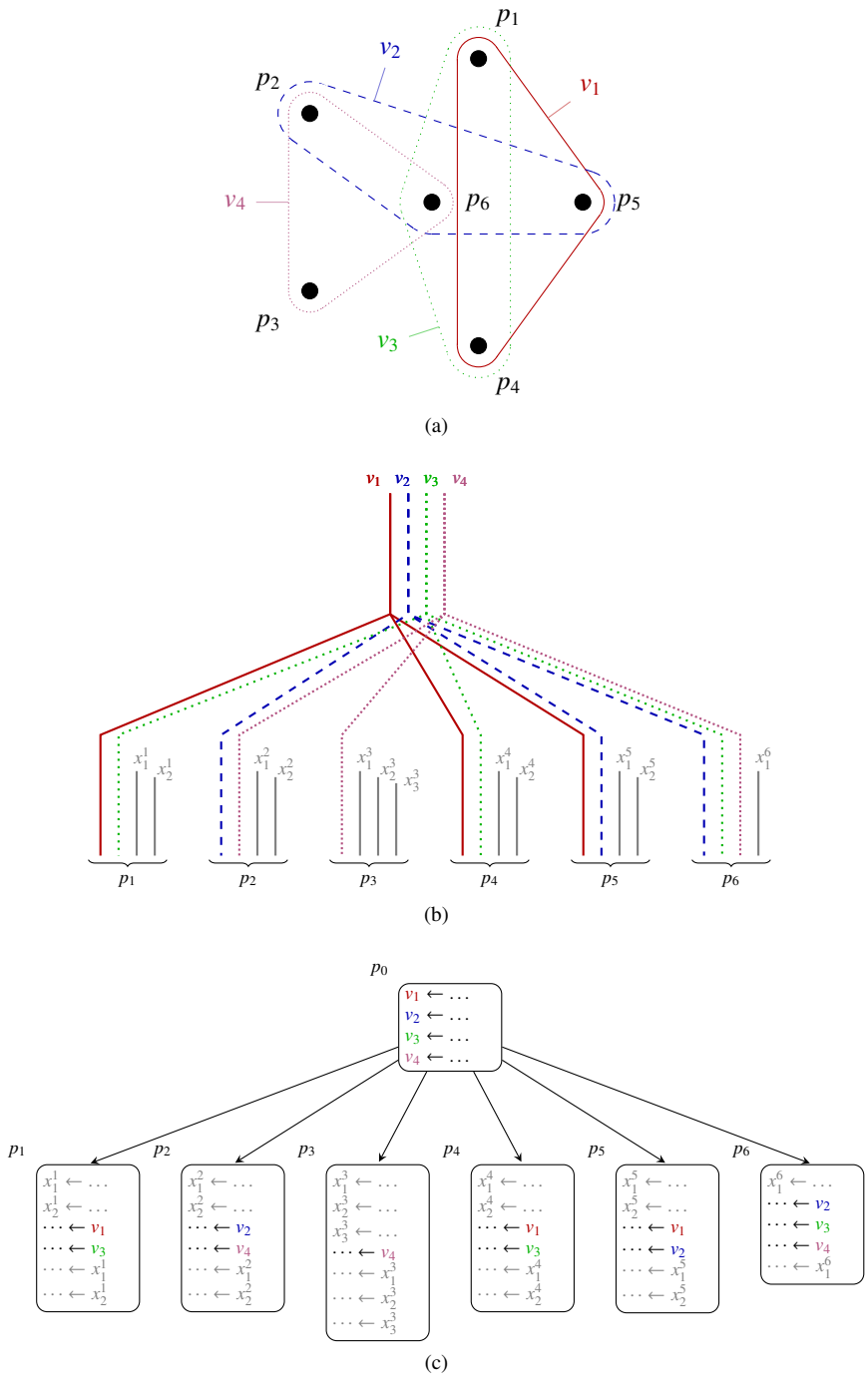


Figure 4.1: Reduction to 3-exact cover: (a) an instance of X3C with $n = 2$ and $m = 4$; (b) corresponding subtrees in the reduction; (c) corresponding SSA code.

(variable) made of the root p_0 and the three points $p_\alpha, p_\beta, p_\gamma$. The number of variables intersecting p_0 is m, so $\Omega = m$. Let us create as many additional variables as necessary (we call them non-labeled variables) so that the number of intersecting variables is exactly Ω for each point of T. In other words, for a leaf p_j that belongs to k subtrees v_i, we create $m - k$ variables, each containing only p_j. Given this family of subtrees of a tree, consider the corresponding intersection graph (which is chordal). We now show that this instance of X3C has a solution if and only if it is possible to remove (spill) at most $n = K$ variables such that, for each point p, the number of remaining intersecting variables is at most $\Omega - 1$. Notice that the reduction is polynomial: the whole number of variables is at most $3n \times m$.

Suppose there is a solution to the incremental spill everywhere problem and let V_S be the set of removed variables with $|V_S| \leq n$. There is no non-labeled variable in V_S because Ω must be decreased in the $3n$ leaves and only a labeled variable goes over three leaves. Hence V_S contains only labeled variables, $|V_S| = n$, and the corresponding set of subsets $\mathcal{S}$ is a covering of $\mathcal{P}$. Conversely, suppose that the X3C instance has a solution $\mathcal{S}$ and let V_S be the set of corresponding subtrees. Since $\mathcal{S}$ is a covering of $\mathcal{P}$, $|\mathcal{S}| = n$ and there is exactly one intersecting set in V_S for each leaf. So the number of remaining intersecting variables is $\Omega - 1$ for each leaf. As for the root p_0, all variables intersect it, so there is at least one (labeled) variable removed and the number of remaining intersecting variables is at most $\Omega - 1$. In other words, V_S is a solution, with $|V_S| \leq n$, to the incremental spill everywhere problem.

This proves that the incremental spill everywhere problem is NP-complete (the fact it belongs to NP is straightforward). □

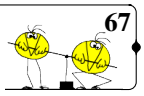
Why is there a difference between this last theorem and Theorem 4.4? In fact, the two problems are not perfectly symmetric: to make a graph k-colorable, the number of kept variables alive at any point should be *at most* k, while to make a graph $\Omega - k$ colorable, the number of removed variables alive at any point must be *at least* k, hence can be arbitrarily large as for the point p_0 in the proof of Theorem 4.6. This is where the combinatorial complexity comes from.

4.2.2 Extension to the spill non-everywhere problem

The spill everywhere problem considers a fixed cost for a live-range. If one want to optimize `loads` and `stores`, i.e., to spill variables only on *parts* of their live-ranges, the cost is not fixed in advance. It is possible to extend the dynamic programming algorithm of Theorem 4.4 to the spill non-everywhere problem. At each point, the cost of spilling a variable now depends on whether it has already been chosen to be spilled by the dynamic algorithm (starting from the leaves)—the cost of the `store` has already been counted—or not, in which case a `store` will be inserted. We will explain our ideas for a basic block, then the dynamic algorithm for a tree, i.e., under SSA, works as in the proof of Theorem 4.4.

The idea for the dynamic programming to work is to separate at each point the variables in three sets:

- the variables not spilled (previously noted W);
- the variables spilled and not in a register (previously noted V_S);
- the variables spilled but still in a register (noted V_S^r).



Indeed, a spilled variable may still reside in a register between two uses, to save one `load`. This complicates the task of calculating the cost of a fitting set. Let us consider a basic block where one tries to calculate the cost of a solution for point p, depending on the solutions for the next point p'.

- If $v \in W(p)$, the only possibility is $v \in W(p')$, with no cost.
- If $v \in V_S(p)$: three cases, $v \in V_S(p')$ with no cost; or $v \in V_S^r(p')$, which costs a `load` (inserted between p and p'); or $v \in W(p')$, which costs a `store` (inserted at the definition of v) and a `load` (inserted between p and p').
- If $v \in V_S^r(p)$: three cases, $v \in V_S^r(p')$ with no cost; or $v \in V_S(p')$ with no cost (v just ceases to reside in a register); or $v \in W(p')$, which costs a `store` (inserted at the definition of v)[7] but no `load` since v is already in a register.

So we just need to be sure that, at each step, there is a polynomial number of $(W(p), V_S(p), V_S^r(p))$ that fit p. Indeed, if C is the number of registers, one needs to choose between $\Omega(p)$ variables those that are in a register (spilled or not): less than $\Omega(p)^C$ possibilities. Then, of all the variables in registers at this point, one needs to choose how many also reside in memory: at most 2^C possibilities. Hence for a fixed C, the number of possibilities is polynomially bounded by $(2\Omega)^C$ and it is easy to use dynamic programming to solve this problem on a basic block.

Under SSA, the dynamic programming is tree-based instead of being linear-based. It works as in the proof of Theorem 4.4. If p has more that one child, a solution for p should match the solutions for all of them, and a special care should be taken when calculating the cost of a solution—if v is spilled in p_1 and p_2, it costs one `store` *less* since both children have already included a `store` for v in their solution cost. Again, it only works because the solutions of children are independent but for some variables that intersect p. When p fixes a "pattern," the choice for children is then independent from other children, and taking the maximum cost gives the best solution.

4.3 Spill Everywhere with Holes

The previous section dealt with the spill everywhere problem without holes. To summarize, by looking again at Table 4.1, this problem is polynomial for a basic block even in its weighted version, whereas it is NP-complete for a general CFG under SSA, unless for a fixed (small) number of registers.

As mentioned earlier, the model without holes does not reflect the reality of most architectures: it corresponds to the CISC-like models while many architectures are in fact RISC-like. The goal of this section is to tackle the problem of spill everywhere "with holes," on a basic block. We restricted the study to the cases where it was polynomial without holes. Indeed, the problem with holes is intuitively "harder" than the problem without holes. Whenever the problem was already NP-complete, chances were that it would stay so.[8]

[7] This is a restriction of the model, where the cost of the `store` is the same everywhere. In practice however, it is better to add it on a place not often executed. It is not trivial to insert this cost in the dynamic programming but it might work.

[8] Actually, this is not straightforward to prove. To patch the proof of Theorem 4.6, other variables must be added to "counter-act" the effects of holes. This is the same technique which will be used to patch the "δ-variables" in the proof of Theorem 4.11.

Where do the holes come from? For an architecture where operations are allowed only between registers, whenever a variable is spilled, one needs to insert a `store` instruction after its definition and `load` instructions before the every use of this variable.[9] Thus, new variables appear, with very short live-ranges, but which nonetheless need to be assigned to registers. In other words, when a variable is spilled, the number of simultaneously alive variables decreases by one at every point of the live-range, *except* where the variable is defined or used. Thus spilling everywhere a variable does not remove the complete interval, but only parts of it, since there is still some tiny sub-intervals left. This is why, for instance, in the algorithm of Chaitin et al. [1981], the register allocation must re-build the interference graph and iterate if some variables are spilled.[10]

Holes and chads: The notion of holes can be formalized as follows. An SSA program on a basic block, or *linear SSA code*, is a pair (B, V) where $B = \{p_1, \ldots, p_m\}$ is a sequence of m program points, and V the set of variables that appear in the code. Between two consecutive program points, there is an instruction.

Each variable of V is defined at most once and, if it is not defined in the code, is considered live-in of the sequence B, i.e., alive on point p_1. Similarly, each variable either has a "last use" (last instruction that uses it) or is live-out of the sequence B. A variable is represented by a simple interval of the sequence B, starting *during* the instruction that defines it (or at p_1 for a live-in), and ending *during* the instruction that last uses it (or at p_m for a live-out). *Spilling* a variable $v \in V$ decreases by one the register pressure at each of its points but not at its definition and uses points, i.e., the program point just after the instruction that defines it, and the program points just before the instructions that use it. Some tiny sub-intervals of the live-range remain at these places. They represent temporary variables that must contain the value in register before storing it or after having reloaded it. Hence, the set of points that is actually "removed" is the interval v with "holes" on it. We call it a *punched interval*. The remaining points $c \in v$ that are not removed are called *chads*, as if, when spilling the variable v, one first had punched the corresponding interval, leaving small intervals in place.

It is important to place precisely where are the holes in live-ranges, since they represent the locations where chads will remain, i.e., where problems will arise. We will do so while referring to Figure 4.2 for a graphical explanation. Note that an instruction *first* uses simultaneously some variables and *then* possibly defines some other new variables. Hence, the holes for the definitions come a bit *later* than the holes for the arguments. This the expected behavior since for instance for $[d \leftarrow a + b]$, if a and d are spilled, the same register can be used to load a and to hold d before its store. Similarly, at a program point, the holes for the definitions of the preceding instruction should not overlap with the holes for the uses of the next instruction. For instance, between the definition of c and the one of d, if c and a are spilled, the same register can be used to store c, then to load a. So, holes for definitions start "in the middle" of the defining instruction and end at the next program point, while holes for uses start at the previous program point and end "in the middle" of the instruction which uses the variables.

[9] We are still in the "spill everywhere" model. For the load/store optimization, not all loads might be required.

[10] Actually, the reason why re-building the interference graph is necessary is more clear in the version of Chaitin [1982]. In the previous version, they reduced the register pressure to k, then tried to color. If it did not work, they would reduce Maxlive to $k-1$, $k-2$,... until it worked. The interference graph must then be built each time before testing if they can color it.

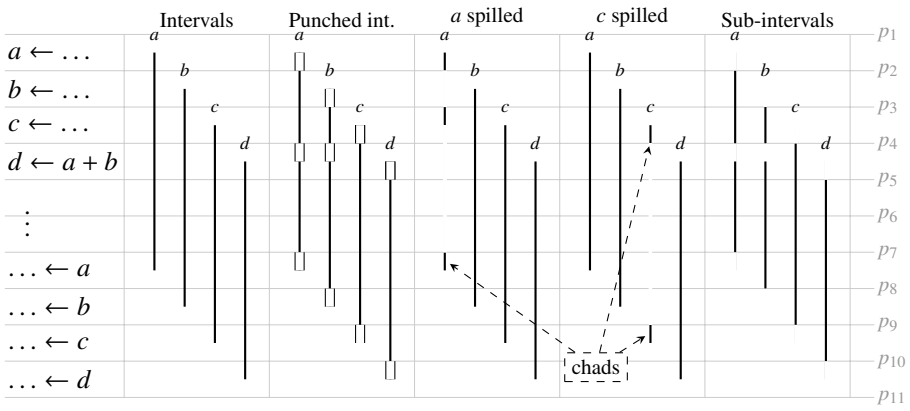


Figure 4.2: Example of punched intervals. When spilled, they leave small intervals (chads) at their definition and uses points. The chads count in the register pressure hence spilling a is not equivalent as spilling c: Maxlive stays at 4 with the former, but drops to 3 with the latter.

Simultaneous holes: Now that we know precisely where are the holes, we can distinguish different cases depending on the *number of simultaneous holes*, written h. This number corresponds to the maximum number of registers that can be used—as arguments—by the same instruction or defined—as results—by the same instruction. It is dependent on the instruction set of the architecture; for instance, $h = 2$ in the three operand addition `add %reg1, %reg2` $\rightarrow$ `%reg3`: it uses two variables at a time, *then* defines one variable.

Live variables: Once some variables V_S have been spilled, the induced code can be characterized as follows. The set of spilled variables alive at p is $V_S(p) = V_S \cap \text{Live}(p)$; the set of non-spilled alive variables is $\text{Live}'(p) = \text{Live}(p) \setminus V_S(p)$. The register pressure after spilling is denoted by $\Omega'(p)$. Notice that $\text{Live}'(p)$ does not contain any chad, whereas of course $\Omega'(p)$ needs to take remaining chads into account. Hence $\Omega'(p)$ is not necessarily equal to $|\text{Live}'(p)|$; more generally, $|\text{Live}'(p)| \leq \Omega'(p) \leq |\text{Live}'(p)| + h$.

All previous notions can be generalized to a general SSA program. The sequence B (linear code) becomes a tree T (dominance tree) and punched intervals become punched subtrees. Now, the (general) problem can be stated as follows.

Problem: SPILL EVERYWHERE WITH HOLES
Instance. A program (T, V) with Maxlive Ω, a weight $w(v) > 0$ for each variable, integers R and K.
Question. Is it possible to spill a set of variables $V_S \subseteq V$ with overall weight $\sum_{v \in V_s} w(v) \leq K$ such that the induced code has Maxlive $\Omega' \leq R$?
Other instances. The spill everywhere *on a basic block* denotes the case where T is a sequence B (linear code). The spill everywhere *with few registers* (C) denotes the case where R is fixed equal to C. The spill everywhere *with many registers* ($\Omega - C$) denotes the case where R is fixed equal to $\Omega - C$. The *incremental* spill everywhere denotes the case where R equals $\Omega - 1$.

As explained by Farach-Colton and Liberatore [2000], the hardness of load-store

	weight	$\Omega' \leq C$	$\Omega' \leq R$	$\Omega' \leq \Omega - C$	$\Omega' \leq \Omega - 1$
$h = 1$	no		?		
	yes		stable set 4.11		
$h \geq 2$	no		stable set 4.10		
	yes			dyn. prog. 4.8	
h not bounded	no				set cover 4.9
	yes	dyn. prog. 4.7			

polynomial NP-complete new result

Table 4.2: Spill on interval graphs (basic blocks) with holes. References to theorems are given in gray. All cases of a colored area are subsumed by the proof given in this area.

optimization for a basic block comes from asymmetry between the cost of the store, which appears only once—when a variable is chosen to be evicted—and the cost of the loads, which is not fixed since it depends on how many times the variable is evicted. Neglecting the cost of the store would lead to a polynomial problem where each sub-intervals of the punched interval could be considered independently for spilling. But we feel that this approximation is not satisfactory in practice because the average number of uses for each variable can be small. Indeed, we measured on our compiler tool-chain, using small kernels representative of embedded applications, that most spilled variables have at most two uses. Hence, minimizing the number of spilled variables is nearly as important as minimizing the number of uses that need a load.

Consider for example a "furthest first"-like strategy on sub-intervals (see Figure 4.2 for an illustration of sub-intervals). To design such a heuristic, a spill everywhere solution might be considered to drive decisions: between several candidates that end the furthest, which one is the most suitable to be evicted in the future? Unfortunately, as summarized by Table 4.2, most instances of spill everywhere with holes are NP-complete for a basic block.

Let us start with a result similar to Theorem 4.4: even with holes, the spill everywhere problem with few registers is polynomial.

Theorem 4.7 (Dynamic programming on non-spilled variables)**.** *The spill everywhere problem with holes and few registers ($R = C$) is polynomially solvable if G is chordal, even in its unweighted version ($w \neq 1$).*

Proof. The proof is similar to the proof of Theorem 4.4. The only point is to adapt the notations to take chads into account. The word "removed" has to be replaced by "spilled" since variables are not removed entirely. Furthermore, the definition of "fitting set" needs to be modified. A set F_p of variables is a fitting set for p if, when all variables not in F_p are spilled, the new register pressure $\Omega'(p)$ is at most C. In other words, the set of fitting sets becomes $\mathcal{F}_p = \left\{\text{Live}'(p);\ \Omega'(p) \leq C\right\}$. Hence, it is "harder" for a set to be a fitting set than for the problem without holes. Therefore, the number of fitting sets is smaller and is still at most $\text{Live}(p)^C \leq \Omega^C$.

As in Theorem 4.4, the proof is an induction on points p of T (from the leaves to the root) and on fitting live sets $F_p \in \mathcal{F}_p$. $W_{max}(p, F_p)$ is built, for each $F_p \in \mathcal{F}_p$, thanks to dynamic programming, by "concatenating" some well chosen $W_{max}(f, F_f)$. Given a child f of p, we select a fitting set $F_f \in \mathcal{F}_f$ that matches F_p, i.e., such that $F_f \cap \text{Live}(p) = F_p \cap \text{Live}(f)$, and that maximizes the cost of $W_{max}(p, F_p)$. This is done for each child of p, and because by construction they match on p, they can be expanded to a solution $W_{max}(p, F_p)$ that fits T_p. The arguments are the same as for Theorem 4.4 and are not repeated here. □

We have seen that, without holes, the spill everywhere problem on an SSA program, with few registers, is polynomial whereas the instance with many registers ($R = \Omega - C$) is NP-complete (Theorem 4.6): the number of spilled variables alive at a given point can be arbitrarily large (up to Ω). For a basic block, this was not the case and we have seen a dynamic algorithm (Theorem 4.5). Now we will see that, if h is fixed, this is still the case. The number of spilled variables is bounded by $2(h + C)$, leading to a dynamic programming algorithm with $O(|B|\Omega^{2(h+C)})$ steps.

Theorem 4.8 (Dynamic programming on spilled variables)**.** *The problem of spill everywhere with holes and many registers ($R = \Omega - C$) can be solved in polynomial time, for a basic block, if h is fixed and even if $w \neq 1$.*

Proof. The key point is first to prove that, for an optimal solution, for each point p, $|V_S(p)| \leq 2(h + C)$. Let us consider a point p such that $|V_S(p)| \geq h + C + 1$, and extend this point to a maximal interval I such that on any point p of this interval, $|V_S(p)| \geq h + C + 1$. We claim that there is no spilled variable $v \in V_S$ completely included in I. Indeed, otherwise, if v were restored (un-spilled) in the final solution, then, at each point p of v, at least $(h + C + 1) - 1 = h + C$ variables would still be spilled, so the register pressure $\Omega'(p) \leq |\text{Live}'(p)| + h \leq (\Omega - (h + C)) + h = \Omega - C$ would still be small enough. This would contradict the optimality of the initial solution. Hence, no variable of V_S is completely included in I: either it starts before the beginning of I, or it ends after the end of I. But I is of maximal size, hence on both extremities, there are at most $h + C$ live spilled variables. Since there is no variable completely included in I, for any point p of I, all variables alive at p goes beyond I either at its start or at its end. This means that there is at most $2(h + C)$ spilled variables alive in any point of I.

The rest of the proof is similar to the proofs of Theorems 4.4 and 4.7. The only difference is that spilled variables are considered instead of kept variables. For a point p, an *extra* live set E_p is a set of variables of cardinal at most $2(h + C)$ and such that, if E_p is spilled, the new register pressure $\Omega'(p)$ becomes lower than R. Let $\mathcal{E}_p$ be the set of extra sets for p. It has at most $\text{Live}(p)^{2(h+C)} \leq \Omega^{2(h+C)}$ elements.

The proof is an induction on points p of $B = \{p_1, \ldots, p_m\}$ and on extra live sets $E_p \in \mathcal{E}_p$. Let $B_{p_i} = \{p_1, \ldots, p_i\}$. A set of variables is said to fit B_p if, for all points in B_p, the register pressure obtained if all other variables are spilled is at most $R = \Omega - C$. The induction hypothesis is that a solution $W_{max}(p, E_p)$ of maximum cost, that fits B_p, and with $V_S(p) = E_p$, can be built in polynomial time. Let p be a point of B and q its predecessor. Consider $E_p \in \mathcal{E}_p$, and an extra live set E_q that matches E_p, i.e., such that $E_q \cap \text{Live}(p) = E_p \cap \text{Live}(q)$, and that maximizes the cost of $W_{max}(q, E_q)$. As noticed earlier, $\left|\mathcal{E}_q\right| \leq \Omega^{2(h+C)}$ and it can be built, by induction hypothesis, in polynomial time. Because E_p and E_q match, $W_{max}(q, E_q)$ can be expanded to a solution $W_{max}(p, E_p)$ that fits B_p. The arguments are the same as those used for Theorems 4.4 and 4.7.

The proof is constructive and provides an algorithm based on dynamic programming with $O(|B|\Omega^{2(h+C)})$ steps. □

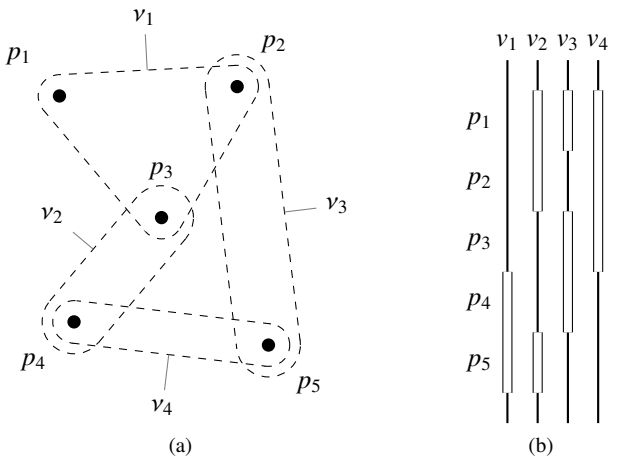


Figure 4.3: Reduction to Set Cover: (a) instance of Set Cover and (b) corresponding punched intervals.

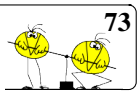
At this point, one might wonder why the dynamic programming of the previous proof might not work for a tree, for instance if the number of children at each branch is bounded. The problem is that the number of leaves of the tree would still not be bounded, and the reduction to 3-exact cover of Theorem 4.6 would still work. The next two theorems show that, as one would expect from the number of steps in the dynamic programming, the complexity does depend on h and C. If $C = 1$, i.e., $R = \Omega - 1$, but h is not fixed, the incremental problem is NP-complete (Theorem 4.9). If h is fixed but there is no constraint on R, most instances are NP-complete (Theorems 4.10 and 4.11).

Theorem 4.9 (From Set Cover)**.** *The incremental spill everywhere with holes is NP-complete if h can be arbitrary, even if $w(v) = 1$ for each $v \in V$ and even on a basic block.*

Proof. The proof is a straightforward reduction from Set Cover [Garey and Johnson, 1979, Minimum Cover, Problem SP5]. Let $\mathcal{V}$ be subsets of a finite set $\mathcal{B}$ and $\mathcal{K} \leq |\mathcal{V}|$ be a positive integer. Does $\mathcal{V}$ contain a cover for $\mathcal{B}$ of size $\mathcal{K}$ or less, i.e., a subset $\mathcal{V}' \subseteq \mathcal{V}$ such that every element of $\mathcal{B}$ belongs to at least one member of $\mathcal{V}'$? Punched intervals can be seen as subsets of B, they contain all the interval points, except chads.

Consider an instance of Minimum Cover. To each element of $\mathcal{B}$ corresponds a point of B. To each element v of $\mathcal{V}$ corresponds a punched interval v that traverses entirely B and that only contains points corresponding to elements of v (see Figure 4.3). In other words, there is a chad for each point *not* in v. Note that to obtain a code with instructions, it is possible to group two consecutive points into one instruction (which uses every variable that has a chad on the first point, and defines every variable that has a chad on the second point), or to utilize instructions that only use variables without defining any or the converse.

At each point p of B, the number of punched intervals and chads that contain p (live variables) is exactly $\Omega = |V|$. A spilling that lowers by at least one the register pressure Ω provides a cover of B and conversely. So, setting $K = \mathcal{K}$ and $R = \Omega - 1$ proves the theorem. □



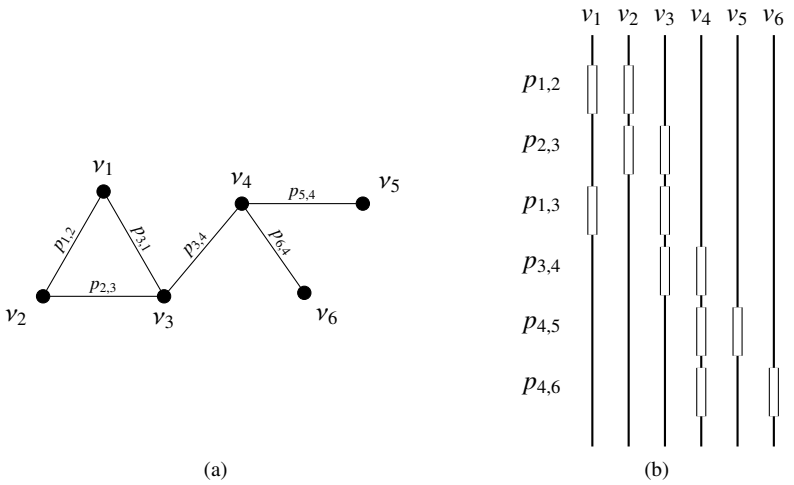


Figure 4.4: Reduction to Independent Set for $h = 2$: (a) instance of Independent Set; (b) corresponding punched intervals.

Notice that the previous proof is very similar to the proof given by Farach-Colton and Liberatore [2000, Lemma 3.1]. Their lemma proves the NP-completeness of the load-store optimization problem, which is harder than our spill everywhere problem. Still, their reduction is similar to ours since they used a trick to force the overall load cost to be the same for all spilled variables, independently from the number of times a variable is evicted. Hence, the optimal solution to their load-store optimization problem just behaves like a spill everywhere solution.

The main limitation of the reduction used for Theorem 4.9 is that the proof needs the number of simultaneous chads h to be arbitrary large, as large as $|V|$. This is of course not realistic for real architectures. Usually, $h = 2$ in practice, and even $h = 1$ for paging problems. Similarly to ours, the reduction of Farach-Colton and Liberatore [2000] use a large amount of simultaneous uses (in their article, a `read` corresponds to a use here and their α corresponds to our h). Their Theorem 3.2 extends their lemma to the case $\alpha = 1$ but, again, it deals with load-store optimization problem, which is harder than spill everywhere. Unfortunately, their trick cannot be applied to prove the NP-completeness of our "simpler" problem and we need to use a different reduction as shown below.

Theorem 4.10 (At most 2 simultaneous chads)**.** *The problem of spill everywhere with holes is NP-complete even if* $w(v) = 1$ *for all* $v \in V$*, even with at most* 2 *simultaneous chads, and even on a basic block.*

Proof. The proof is a straightforward reduction from Independent Set (also called Stable Set) [Garey and Johnson, 1979, Problem GT20]. Let $G = (\mathcal{V}, E)$ be a graph and $\mathcal{K} \leq |\mathcal{V}|$ be a positive integer. Does G contain an independent set (stable) $\mathcal{V}_S$ of size $\mathcal{K}$ or more, i.e., a subset $\mathcal{V}_S \subseteq \mathcal{V}$ such that $|\mathcal{V}_S| \geq \mathcal{K}$ and no two vertices in $\mathcal{V}_S$ are joined by an edge (adjacent) in E?

Consider an instance of Independent Set. To each vertex $v \in \mathcal{V}$ of G corresponds a variable $v \in V$ that is alive from the entry of B to its exit. To each edge $(v, v) \in E$ of G

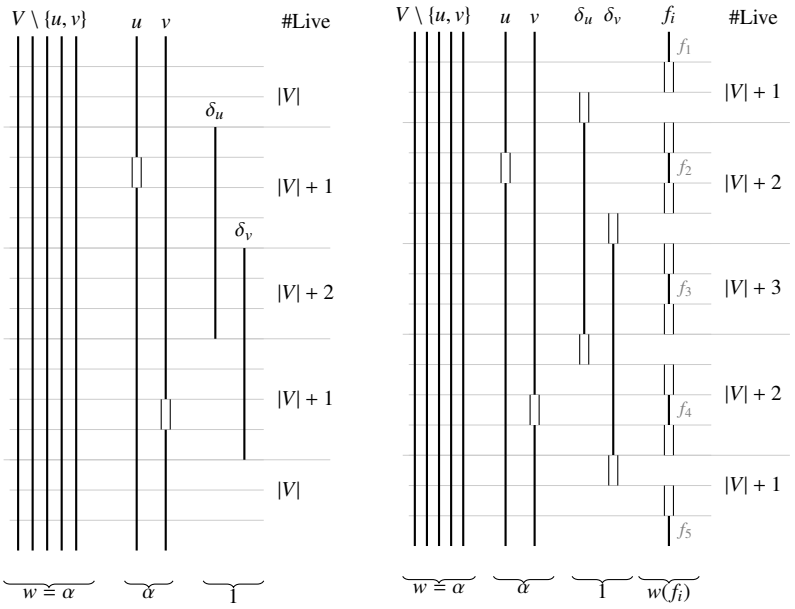


Figure 4.5: Reduction from Independent Set for $h = 1$: (a) region for an edge (u, v); (b) actual code must have holes at extremities of δ variables.

corresponds a point $p_{u,v}$ of B that contains a use of the corresponding variables u and v (see Figure 4.4). In other words, there are two chads for each point of B. The key point is to notice that spilling K variables in V_S lowers Ω to $|V| - K + 1$ if and only if the corresponding set of vertices $\mathcal{V}_S$ is an independent set. Indeed, if $\mathcal{V}_S$ contains two adjacent vertices u and v, then at point $p_{u,v}$, the register pressure would be $|V| - K + 2$. Hence, by setting $K = \mathcal{K}$ and $R = |V| - K + 1$, we get the desired reduction. Indeed, if there exist $k \leq K$ variables that, when spilled, lead to a register pressure at most $R = |V| - K + 1$ then: first, k must be equal to K, second, the corresponding vertices form an independent set of size K. Conversely, if there is an independent set of size at least K, then spilling the corresponding variables leads to a register pressure at most $|V| - K + 1$. □

Now that we have seen the case when $h = 2$, the reader should be mature enough to enter the case $h = 1$. The reduction is similar to the one in the above proof, from Independent Set. But in this proof we strongly used that two chads might be simultaneous. In the proof of the following theorem, we will use the same structure, but slightly "displace" the chads so that they are not simultaneous anymore. The trick is then to add many other small variables, and we will see that it "all fit together." However, we now need to use weights to distinguish the small variables from the main ones.

Theorem 4.11 (No simultaneous chads)**.** *The spill everywhere problem with holes is NP-complete even if $h = 1$ and for a basic block, in its weighted version ($w \neq 1$).*

Proof. As for Theorem 4.10, the proof is a reduction from Independent Set. Consider an instance of Independent Set. To each vertex $v \in \mathcal{V}$ of G corresponds a variable $v \in V$

(called vertex variables), which is alive from the entry of B to its exit. To each edge $(v, v) \in E$ of G corresponds a region in B where u and v are consecutively used. For our needs, as depicted in Figure 4.5a, we add two overlapping local variables, δ_u and δ_v, called δ variables. In real codes, every live-range must contain a chad at the beginning and a chad at the end. For our proof, we need to be able to remove the complete live-range of a δ variable, which is of course not possible because of the presence of chads for such variables. To avoid this problem, we will introduce later new "f_i" variables to increase the register pressure by one everywhere except where δ variables have chads. But now, for the sake of clarity, we will still consider that δ live-ranges contain no chads and delay the appropriate corrections to the end of the proof.

Let us choose $K = \mathcal{K}$ and $R = |\mathcal{V}| - K + 1$. The cost for spilling a vertex variable will be α while the cost for spilling a δ variable will be 1. As for α, the trick is to make sure that an optimal solution of our spilling problem spills exactly K vertex variables and $|E|$ of the δ variables (i.e., at least one per region, and exactly one is sufficient), for a cost equal to $\alpha K + |E|$. This is ensured by setting $\alpha = 2|E| + 1$.[11] Indeed: first, spilling $K - 1$ vertex variables even with all the δ variables is not enough: on a chad of any of the spilled variables,[12] the register pressure would be lowered to $|\mathcal{V}| - (K - 1) + 1 = |\mathcal{V}| - K + 2 > R$. Second, spilling K vertex variables requires to spill at least one δ variable per region and spilling all δ variables is enough. Hence, the minimum cost of a spilling with exactly K vertex variables is between $K\alpha + E$ and $K\alpha + 2E$. Finally, spilling $K + 1$ vertex variables—and no δ variable—has a cost equal to $(K + 1)\alpha = K\alpha + 2|E| + 1$.

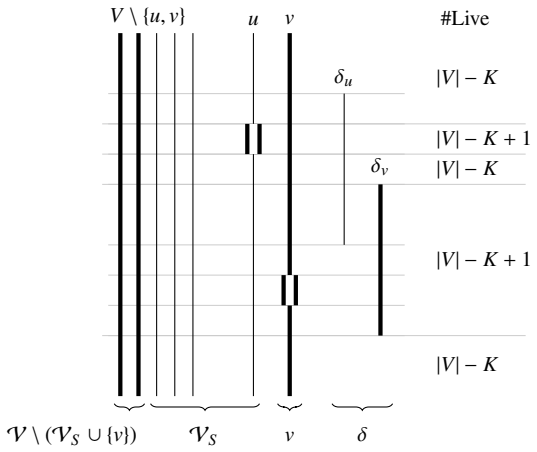
Now, it remains to show that the cost of an optimal spilling is $K\alpha + E$ if and only if (iff) the spilled variables define an independent set for G. All situations for an edge (u, v) are depicted in Figure 4.6. If both u and v are spilled—in which case $\mathcal{V}$ is not a stable set—then both δ_u and δ_v must be spilled and the cost cannot be $K\alpha + E$. Otherwise, spilling either δ_u (if u is spilled) or δ_v (if v is spilled) is enough.

To finish this proof, we need to get back to the problem that, in fact, the δ variables have two chads: one for their definition at the beginning of their live-range, and one at the end for their last use. We patch the proof by adding five new variables $(f_i)_{1 \leq i \leq 5}$ for every edge (u, v) in G as depicted on Figure 4.5b. In fact, the f_5 for an edge can be the f_1 of the next edge in the reduction. The goal is that the union of the live-ranges of f_i variables covers exactly all points of B, except the points that correspond to the chad of a δ variable. This makes a total of $4|E| + 1$ new f_i variables. The cost $w(f_i)$ of spilling a variable f_i is then chosen large enough so that no f_i variable will ever be spilled in an optimal solution, and one more register is provided for these variables in the reduction. □

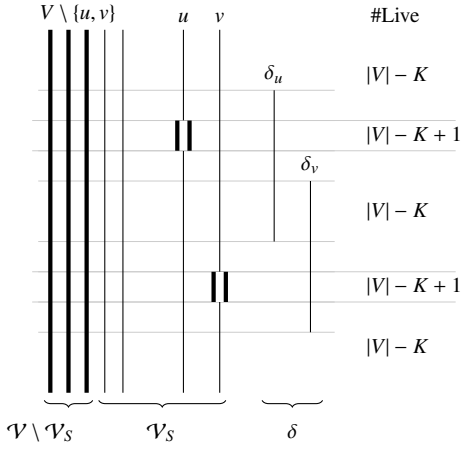
Finally, only one problem remains. What about the spill everywhere problem with holes when $h = 1$ in its unweighted version? We did not came up with arguments toward the polynomial or NP-completeness direction, so this is still an open problem. This problem is nevertheless even farther from architectural realities than previous problems—one should have an architecture that can read or write only one register at a time *and* a mechanism such that no matter how many times you reload a value, it still costs the same (to get $w = 1$). We thought that there is already plenty of clues that the spilling problem is a difficult one, even under SSA or for basic blocks, in its

[11] $\alpha = |E| + 1$ would be enough but would complicate the proof.

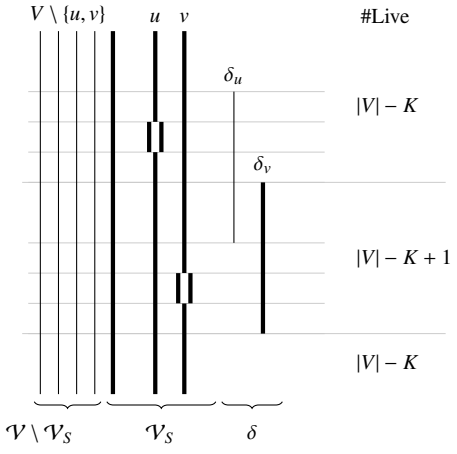
[12] We can safely suppose there is no isolated node in G, otherwise, such nodes can always be in a stable set, hence they can be removed from G and $\mathcal{K}$ decreased by their number.



(a)

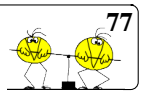


(b)



(c)

Figure 4.6: Different configurations for the reduction with $h = 1$: (a) only u is spilled, (b) both u and v are spilled, (c) none of them are spilled. Non-spilled variables are in bold and $R = |\mathcal{V}| - K + 1$ registers.



simpliest "spill everywhere" version. So, answering this last problem would probably not help in the design of spilling heuristics.

4.4 Conclusion

The recent result that, under the SSA form, the interference graph of a program is chordal opened promising directions for the design of register allocation heuristics, by having an exact test to decide whenever spilling is necessary, and a polynomial algorithm to assign registers to variables when no spilling is necessary anymore. Studying the complexity of the spill "everywhere" problem—where variables are spilled on their entire live-range—was important in this context. Even if it is a restriction of the more general load-store optimization problem, the "everywhere" simplification is used by many register allocators (e.g. Iterated Register Coalescing (IRC)), and might either give clues for the load-store optimization problem, or work as an oracle where the spill everywhere approximation is sufficient, in JIT compilation for instance.

Our results can provide insights for the design of aggressive register allocators that trade compile time for provably "optimal" results. But, unfortunately, the main implication of our work is that SSA does not simplify the spill problem like it does for the assignment (coloring) problem. Our study considers different singular variants of the spill everywhere problem:

1. We distinguish the problem without or with holes depending on whether use operands of instructions can reside in memory slots or not. Live-ranges are then contiguous or with holes, which leaves chads when spilled.

2. For the variant with chads, we study the influence of the number of simultaneous chads—maximum number of use operands of an instruction and maximum number of definition operands of an instruction.

3. We distinguish the case of a basic block (linear sequence) and of a general SSA program (tree).

4. Our model uses a cost function for spilling a variable. We distinguish whether this cost function is uniform (unweighted) or arbitrary (weighted).

5. Finally, in addition to the general case, we consider the singular case of spilling with few registers and the case of an incremental spilling that would lower the register pressure one by one.

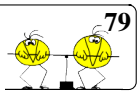
The classical furthest-first greedy algorithm is optimal only for the unweighted version without holes on a basic block. The weighted version can be solved in polynomial time, but unfortunately only for a basic block and not for a general SSA program. The positive result of our study for architectures with few registers is that the spill everywhere problem with a bounded number of registers is polynomial even with holes. Of course, the complexity is exponential in the number of registers, but for architectures like X86, it points that algorithms based on dynamic programming might be considered in an aggressive compilation context. In particular, it may be a possible alternative to commercial solvers required by ILP formulations of the same problem, even for models more general than spill everywhere as the one used by Appel and George [2001]. However, ILP is often faster than dynamic programming.

For architectures with a large number of registers, we studied the *a priori* symmetric problem where one needs to decrease the register pressure by a constant number; our hope was to design a heuristic that would incrementally lower one by one the register pressure to meet the number of registers. Unfortunately, it is NP-complete to decrement the register pressure even by one.

Our study also shows that the complexity also comes from the presence of chads. The problem of spill everywhere with chads is NP-complete even on a basic block. On the other hand, the incremental spilling problem is still polynomial on a basic block provided that the number of simultaneous chads is bounded. Fortunately, this number is very low on most architectures.

To conclude, our results for the spill problem do not match our expectations. While we hoped to find that SSA simplifies the problem, we were in fact confronted mainly by NP-complete problems. What good might we take out of this theoretical study? Although we did not dig up any useful or stunning concept, we hope that our proofs correctly point to where the complexity really is. For instance, we created trees with an enormous amount of leaves in Theorem 4.6, or used many times the same variables in instructions in Theorem 4.10. Maybe these clues can be exploited to improve the most promising spilling heuristics, for instance a Belady-like algorithm as do Hack [2007].

More importantly, the discovery that the interference graph of a program under SSA form is chordal led to the writing of many articles with sometimes misleading titles. For instance, Hack and Goos [2006] published "Optimal register allocation for SSA-form programs in polynomial time," and Brisk et al. [2005] "Polynomial time graph coloring register allocation." Our study does not invalidate these articles; it is true that SSA simplifies the problem of coloring the interference graph to the point where an optimal coloring is found in polynomial time. We will also see in Chapter 5 that some coalescing problems are simpler on chordal graphs. However, the titles of these articles ignore the simple fact that coloring is *not* allocation, and our study shows that we must not forget where the complexity of register allocation really comes from: the spilling problem. In this context, it was mandatory to do a thorough study of the spilling problem under SSA form to see if that would simplify the problem. Unfortunately, it does not.



Do not mistake coalescing for koalescing (main activity of koalas).

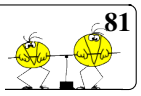
5

On the complexity of register coalescing

The complexity of register allocation for a fixed schedule comes from two main optimizations, *spilling* and *coalescing*, as we explained in Chapter 3. Spilling decides which variables should be stored in memory so as to make register assignment possible while minimizing the cost of stores and loads. Its complexity for programs under Static Single Assignment (SSA) form was studied in Chapter 4. Register coalescing reduces register-to-register moves by allocating preferably two variables involved in a move instruction to the same register. This chapter and the next one are devoted to the study of coalescing problems. In this chapter, we study the complexity of coalescing problems, while, in Chapter 6, we will propose practical coalescing schemes and compare them to existing strategies.

We presented in Chapter 2, Section 2.2.4, the Iterated Register Coalescing (IRC) scheme introduced by George and Appel [1996]. This graph-based register allocation scheme is now very popular due to its clean and reproducible design. But, with the increasing need for optimizing memory transfers, either for performance or power consumption, it is important today to find heuristics that spill less, possibly at the price of additional register-to-register moves. Several variants have been proposed to avoid, as much as possible, these additional moves. *Aggressive coalescing*, or "reckless coalescing," was in fact already present in the first algorithm of Chaitin et al. [1981]. It made more sense at this time because the machine they were working with (an IBM 801) had a zero-wait-state memory, so moves and spill instructions had equal cost. Aggressive coalescing effectively removed many copies but was abandoned in later improvements because it produced too many spills. It then reappeared later as the first phase of the *optimistic coalescing* of Park and Moon [1998, 2004]. Their idea is to perform first an aggressive coalescing, optimistically, i.e., hoping not to generate too many nodes hard to color. Then, during the "select" (coloring) phase, instead of spilling right away nodes that cannot be colored, they split them back into their initial components, i.e., *de-coalesce* them.

New coalescing problems have also appeared due to recent developments on SSA form. Today, most compilers go through this intermediate code representation that makes many code optimizations simpler. Under SSA, each variable is defined textually only once, but the ϕ-functions used to emulate the transfer of values at join points of the control-flow graph (CFG) are not machine code. When going back to ordinary code, an out-of-SSA phase is necessary, which typically introduces many register-to-register moves. Several techniques are available to go out of SSA [Cytron et al., 1991; Briggs et al., 1998; Leung and George, 1999a; Sreedhar et al., 1999; Budimlić et al., 2002; Rastello et al., 2004], some with the objective of reducing the number of moves.



This problem is a form of aggressive coalescing as no register constraint is taken into account in this phase, which is done *before* register allocation. With an adequate interpretation of ϕ-functions, this is an aggressive coalescing problem performed on special graphs as interference graphs of SSA programs are chordal.

Our experiments with classical out-of-SSA approaches revealed many bad situations where a too aggressive coalescing can increase the number of spills in the subsequent register allocation phase. Some splitting is then needed to undo the coalescing, but this is difficult to control. Also, a standard conservative coalescing approach is sometimes not enough to coalesce most copies that arise in the out-of-SSA phase, in particular copies corresponding to permutations. Thus, the interplay between register allocation, out-of-SSA approaches, and register coalescing needs to be clarified.

Finally, the fact that the interference graph of a program under SSA is chordal, and therefore easy to color, has also led to the developments of new heuristics for register allocation, based on two separate phases, one for spilling and one for coalescing, which is what we advocate in this thesis. The first phase of spilling decides which values are spilled and where, so as to get a code with Maxlive $\leq R$, the number of registers. The second phase of coloring, the register assignment, maps variables to registers with no additional spill, although we have seen in Chapter 3 that this is more subtle than this quick explanation. When possible, it also removes copy instructions—also called "shuffle code" by Lueh et al. [2000]—thanks to coalescing. Other people advocate this approach, for instance Appel and George [2001] and, more recently, Brisk et al. [2005] and Hack et al. [2006]. The coalescing phase of such an approach seems *a priori* simpler than for Chaitin-like register allocators because we already know how to color the initial graph with R colors. One just wants to coalesce as many moves as possible so that the graph remains R-colorable or, more precisely, easy to color with R colors. However, the fact that the first phase of spilling can be much more aggressive makes the coalescing more difficult. After spilling just the necessary variables, the code may have a very high register pressure, possibly equal to Maxlive at many program points, and many moves corresponding to permutations of R colors. To coalesce such moves, standard conservative coalescing approaches are not effective enough. This led Appel and George [2000] to define a "coalescing challenge."

We believe that these new developments and variants of the coalescing problem motivate the need for a better study of its complexity, which has not been addressed in details so far. Indeed, since the NP-completeness proof of Chaitin et al. [1981], the impression was that all the register allocation process was NP-complete, hence nobody was actually interested in studying the NP-completeness of its subproblems. In this chapter, we distinguish the following different coalescing optimizations:

a) *aggressive coalescing* removes as many copies as possible, regardless of the colorability of the resulting interference graph;

b) *conservative coalescing* removes as many copies as possible while keeping the colorability of the graph;

c) *incremental conservative coalescing* removes one particular copy while keeping the colorability of the graph;

d) *optimistic coalescing* coalesces all copies aggressively, then gives up on as few copies as possible so that the graph becomes colorable again.

We (almost) completely classify the complexity of these problems, considering also the structure of the interference graph: arbitrary, chordal, or greedy-k-colorable

(see definitions in Chapter 2, Section 2.2.2). We view this study as a necessary step for designing new coalescing strategies, which would better exploit the structure of the graphs.

5.1 Definitions & properties for NP-completeness

We introduced the notion of coalescing in Chapter 2, Section 2.2.3, as a way to help the coloring scheme of Chaitin et al. In practice the main advantage of coalescing is to remove unnecessary copies in the program, by merging the corresponding nodes. This defines a notion of "preference" between edges, in which two variables are linked by an *affinity* if assigning the same color to both of them would save some copy instructions during the execution of the program (see Definition 2.12). Affinities can be weighted, in which case the weight represents the gain obtained by coalescing the two variables. For our purpose, we need to define properly the term "coalescing."

Definition 5.1. A *coalescing* of $G = (V, E)$ with affinities $\mathcal{A}$ is a function f such that $f(u) \neq f(v)$ whenever $(u, v) \in E$; an affinity $\langle u, v\rangle \in \mathcal{A}$ is *coalesced* if $f(u) = f(v)$.

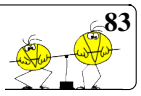
The function f in this definition looks very much like the *col* function of Definition 2.14, i.e., a coloring of the interference graph. Indeed, both share the property that they must assign a different value—i.e., color—to adjacent nodes in the graph. The differences are that the "goal" of *col* is to take at most k different colors to be a k-coloring, while the "goal" of f is to give the same value to affinity neighbors as much as possible. In particular, any k-coloring defines a coalescing—all nodes with the same color are coalesced together—but the converse is false since a coalescing does not have any constraint on the number of colors.

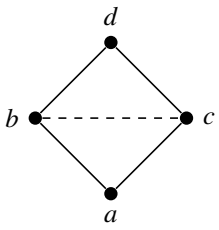
Definition 5.2. The *coalesced graph* $G_f = (V_f, E_f)$ is the graph obtained from G by merging all vertices with the same image under f. More formally, if f takes n values, f defines a partition of V into n subsets $(S_i)_{1\leq i\leq n}$ where u and v are in the same subset if and only if (iff) $f(u) = f(v)$. The vertices in V_f are the subsets $(S_i)_{1\leq i\leq n}$ and there is an edge $(S_i, S_j) \in E_f$ iff $(u, v) \in E$ for some $u \in S_i$ and $v \in S_j$. Since f is a coloring, it is guaranteed that G_f has no self-edge (S_i, S_i).

In the next sections, we prove the NP-completeness of different coalescing problems for particular *interference graphs and affinities*. To prove that the corresponding coalescing problems for *programs* are also NP-complete, we need a way to build, for each graph and set of affinities we consider, a program with interferences and move instructions that is as hard to coalesce. The following property gives such a construction; this will allow us to forget about programs in the next sections and deal with graphs and affinities only.

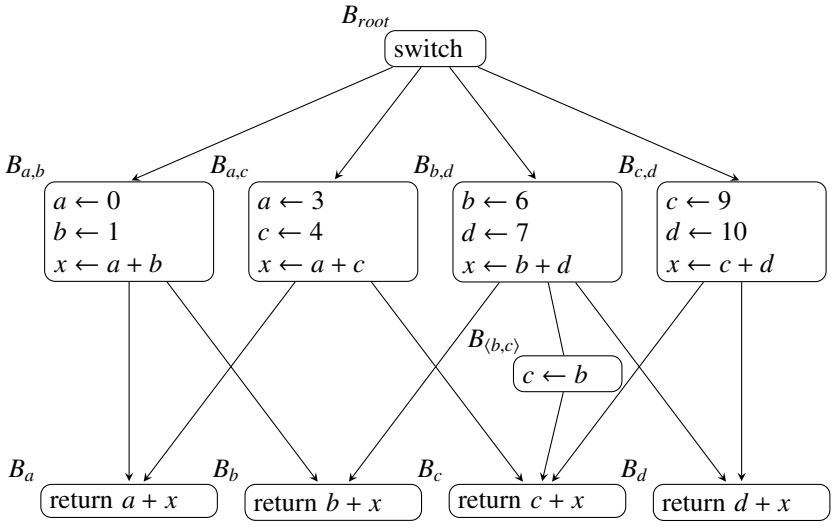
Property 5.3 (Graph and program equivalence)**.** *Let G be a graph and $\mathcal{A}$ a set of affinities. There is a program whose interference graph G' and set of moves $\mathcal{A}'$ is as hard to coalesce. Furthermore, if G is chordal, the program can be chosen in SSA form with dominance property.*

Proof. We will construct two programs, one whenever G is arbitrary and whenever G is chordal. For the arbitrary graph G, the construction of Chaitin et al. [1981] can be used to build a program whose interference graph is as difficult to color as G (see details in Chapter 3). Then, for each affinity $\langle u, v\rangle \in \mathcal{A}$, a block $B_{\langle u,v\rangle}$ is created, with a move





(a)



(b)

Figure 5.1: Chaitin-like reduction with affinities: for any graph and set of affinities (a), there exists a program (b).

instruction $[v \leftarrow u]$ and a control-flow edge from a block where u is live to a block that uses v as shown on Figure 5.1.

For the case when $G = (V, E)$ is chordal (see Figure 5.2a), there is a set of subtrees $(t_v)_{v \in V}$ of a tree T whose intersection graph is G [Golumbic, 1980, Thm. 4.8]. One can also assume that only one subtree starts at a time. Define an orientation on T to get a directed tree and let r be its root (see Figure 5.2c). By a depth-first traversal of T from r, a strict SSA program is deduced (see Figure 5.2d). T is viewed as the CFG of the program[1] and the start (resp. the end) of a subtree T_v is viewed as a definition (resp. a use) of a variable v. The live-ranges of the variables are exactly the subtrees and their intersection graph is G.

It remains to define some move or ϕ instructions corresponding to the affinities. For each affinity $\langle u, v \rangle$, define a new basic block $B_{\langle u,v \rangle}$ and two control-flow edges leading to $B_{\langle u,v \rangle}$, one from the basic block where u is defined, one from the basic block where v is defined. Since these new edges should not extend the live-ranges of u and v inside their definition basic block, it is safer to split the basic blocks just after the definition of u and v so that the new control-flow edges are not added at the end of the blocks. Finally, $B_{\langle u,v \rangle}$ contains the ϕ-function $a_{u,v} = \phi(u, v)$ where $a_{u,v}$ is a new variable. Figure 5.2d show an example of such an SSA code. The complexity of coalescing these ϕ-functions is the same as coalescing the affinities $\mathcal{A}$ with the graph G (see Figure 5.2b). Indeed, for any affinity $\langle u, v \rangle$, one can always coalesce one of the two affinities $\langle u, a_{u,v} \rangle$ or $\langle v, a_{u,v} \rangle$ defined by the ϕ-function. Then, coalescing the remaining affinities is exactly coalescing the affinities $\mathcal{A}$ in the graph G. □

In the next sections, we show several NP-completeness results, for a fixed k, which is stronger than assuming that k is an input of the problem. However, one could wonder if the problem remains NP-complete for another fixed $k' \geq k$. The following property (with $p = k' - k$) will extend our NP-completeness results from k to k'.

Property 5.4 (NP-complete problems are not easier with more registers)**.** *Let G be a graph. Define G' by adding to G a clique of p new vertices and edges between each vertex of the clique and each vertex of G. Then G is k-colorable iff G' is $(k + p)$-colorable, G is chordal iff G' is chordal, and G is greedy-k-colorable iff G' is greedy-$(k + p)$-colorable.*

Proof. The first property is obvious: by construction, the additional clique must use p other colors. For the second property, if G' is chordal, G is also chordal as a subgraph of G'. Conversely, if G is chordal, consider a cycle of G' of length at least 4. If it is a cycle of G, it has a chord. Otherwise, it has a vertex v in the clique and two edges (v, u) and (u, w) with $w \neq v$. Since v is connected to any other vertex in G', (v, w) is a chord. For the third property, suppose that G is greedy-k-colorable, i.e., vertices can be removed in some order, with degree $< k$ in the remaining graph. In G', one can first remove the vertices of G in the same order, as their degree is at most $(k - 1 + p)$. Then one can remove the vertices of the clique, whose degree is $< p$, and thus G' is greedy-$(p + k)$-colorable. Finally, if G is not greedy-k-colorable, it has a subgraph H such that all vertices have degree (in H) at least k. Adding the clique C of size p to H shows that G' is not greedy-$(p + k)$-colorable, because, in H, all vertices have degree $\geq k + p$ and in C, they have degree $\geq p - 1 + |H| \geq p - 1 + k + 1$. □

[1] In our construction, the dominance tree follows the CFG but this is not the case for an arbitrary SSA program.

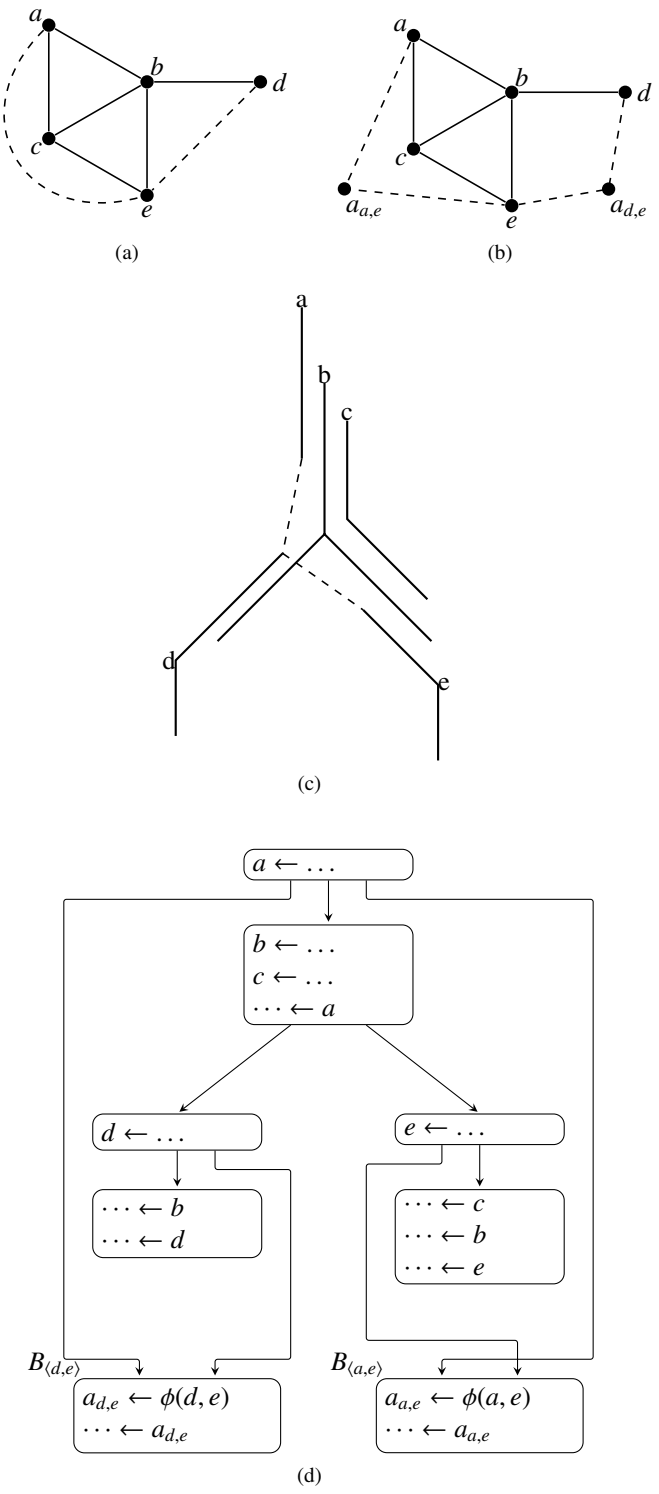


Figure 5.2: Chordal reduction with affinities: for any chordal graph and set of affinities there exists an SSA program.

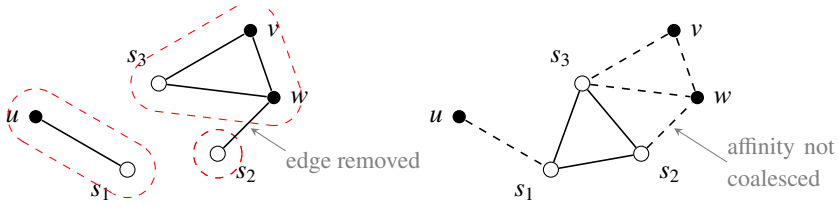


Figure 5.3: Aggressive coalescing: reduction from Multiway Cut.

5.2 Complexity of aggressive coalescing

The goal of the *aggressive coalescing* is to remove as many move instructions as possible, with no constraint on the number of registers. Only interferences can prevent coalescing. In Chaitin et al.'s original algorithm, coalescing was done aggressively before coloring. Also, the first phase of "optimistic" algorithms, like the one of Park and Moon [2004], is an aggressive phase. The problem can be formulated as follows:

Problem: AGGRESSIVE COALESCING
Instance. Graph $G = (V, E)$, affinities $\mathcal{A} \subseteq V^2$, integer K.
Question. Is there a coalescing of G, i.e., a function f with $f(u) \neq f(v)$ whenever $(u, v) \in E$, such that at most K affinities $\langle u, v\rangle \in \mathcal{A}$ are not coalesced, i.e., satisfy $f(u) \neq f(v)$?

Our reduction is from Multiway Cut [Dahlhaus et al., 1992]: given a graph $G = (V, E)$, a subset $S = \{s_1, \ldots, s_k\}$ of V with k specified vertices or *terminals*, an integer K, the problem is to decide if one can remove at most K edges from E so that each terminal is in a different connected component. In the general Multiway Cut problem, edges are weighted but it is NP-complete even for the previous version where all edges have equal weight, and even with only three terminals ($k = 3$).[2]

Theorem 5.5. *The aggressive coalescing problem is NP-complete even if there are only* 3 *interferences.*

Proof. Our reduction is as follows. Let $H = (V, E)$, S, K, be an instance of Multiway Cut. Let us construct the interference graph $G = (V, S \times S)$, i.e., the k terminals (nodes in S) form a clique, and all other nodes have degree zero, i.e., all $v \in V \setminus S$ are isolated vertices. For each original edge $(u, v) \in E$, let us create an affinity $\langle u, v\rangle$ in our interference graph, i.e., $\mathcal{A} = E$. Then, any maximal coalescing defines an k-partition of the nodes. Indeed, any node not in S can be coalesced with one of the terminals, but no two terminals can be coalesced since they are neighbors. Moreover, for each affinity $\langle u, v\rangle$, if u and v are coalesced to different terminals, the affinity is constrained and cannot be coalesced. So the set of affinities not coalesced defines a k-partition of G, and the corresponding set of edges in E defines a k-cut of H. The converse is also true. Figure 5.3 gives an example of this reduction. To conclude, there is a coalescing of G in which at most K affinities are not coalesced iff at most K edges must be removed from E to disconnect the terminals in H. □

[2] Unless G is planar, but this is not of our concern here.

Going out of SSA while minimizing the number of moves is a form of aggressive coalescing. Other proofs related to aggressive coalescing and out-of-SSA translation are available in Rastello et al. [2005]; Hack et al. [2005]. From a complexity point of view, Theorem 5.5 shows that aggressive coalescing is difficult even if the interference graph is very simple, in particular even if it is chordal or greedy-k-colorable. These properties do not make the problem simpler, as, as shown by our reduction, the complexity comes from the *structure of the affinities* and not the one of the interferences. From a practical point of view, aggressive coalescing can degrade register allocation. Indeed, coalescing means fusing live-ranges and merging, in the interference graph G, the corresponding vertices. After these merges, the coalesced graph G_f may not be k-colorable. In this case, three alternatives are available:

- One can remove some vertices from the graph and spill the corresponding variables; this is the strategy proposed by Chaitin [1982] in its register allocator;
- One can give up on some coalesced moves and de-coalesce them so that the graph gets greedy-k-colorable again; this is the strategy of optimistic coalescing [Park and Moon, 1998, 2004] that we analyze in Section 5.4;
- One can prefer to not use aggressive coalescing but to coalesce moves only if the graph is proved to remain greedy-k-colorable; this is conservative coalescing, introduced by Briggs [1992], a technique we analyze in Section 5.3.

5.3 Complexity of conservative coalescing

The *conservative coalescing* problem, for a k-colorable graph, is to coalesce as many moves as possible so that the interference graph remains k-colorable after the coalescing. Another possible formulation by Appel and George [2001] is to ask directly for a coalescing f that is a k-coloring of G. We prefer the first formulation as given below: it is closer in spirit to what heuristics do and it allows us to discuss more precisely the complexity of the problem in terms of the structure of G and G_f. Indeed, with no constraints on G and G_f, the problem is obviously NP-complete: for $\mathcal{A} = \emptyset$ and $K = 0$, this is nothing but Graph k-Colorability [Garey and Johnson, 1979, Problem GT4]. However, the problem may seem simpler in practice, when working on some graph G with a particular structure or colorability, or if one is allowed to merge only vertices connected by an affinity, or if one requires the graph G_f to be not only k-colorable, but also greedy-k-colorable. We will study in details these different cases, as they might give some leads to polynomial algorithms or heuristics likely to work.

Problem: CONSERVATIVE COALESCING
Instance. Graph $G = (V, E)$, affinities $\mathcal{A}$, integers K and k.
Question. Is there a coalescing f of G such that the coalesced graph G_f is k-colorable and at most K affinities are not coalesced?

Theorem 5.6 addresses the complexity of conservative coalescing. Although Appel and George [2001] already proposed a reduction from Graph k-Colorability, they did not give the proof. Here, we will give this reduction to show how to extend their remark into a more accurate complexity result. For a quicker way to show Theorem 5.6 without the accuracy on G and G_f, it is possible to use the proof of Theorem 5.5, since the graph used in this proof is a triangle plus some isolated vertices. It keeps such a structure after any coalescing, thus it is chordal and greedy-3-colorable. Because we wanted a better result in terms of G and G_f, we provide a longer proof.

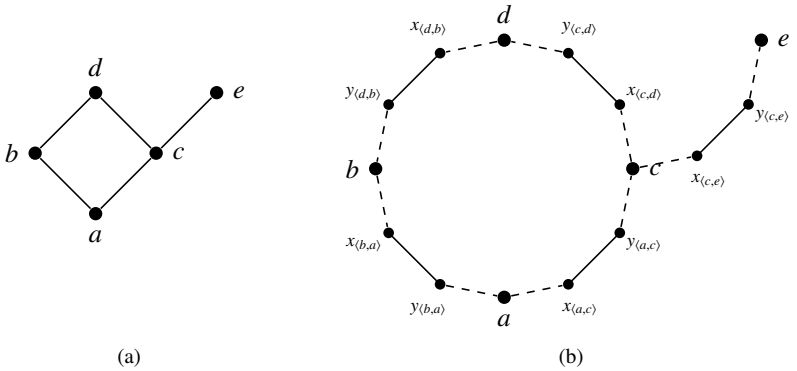


Figure 5.4: Reduction for Thm. 5.6 (first part).

Theorem 5.6. *Conservative coalescing is NP-complete, even for $k = 3$, even if G_f is required to be also chordal or greedy-3-colorable, even if G_f needs to be obtained by merging only vertices connected by affinities, and even if G is greedy-2-colorable.*

Proof. As noticed by Appel and George [2001], a reduction from Graph k-Colorability [Garey and Johnson, 1979, Problem GT4] shows that, even for $K = 0$, the conservative coalescing problem is NP-complete. Indeed, let $H = (V, E)$ be an instance of Graph k-Colorability. Define an instance (G, A, K) of conservative coalescing as follows. The vertices of the interference graph G are the vertices of H plus some new vertices, two vertices x_e and y_e for each edge $(u, v) \in E$. The interferences in G are the pairs (x_e, y_e) and the affinities in $\mathcal{A}$ the pairs $\langle u, x_e\rangle$ and $\langle v, y_e\rangle$ (see Figure 5.4). All moves can be aggressively coalesced and the coalesced graph G_f is thus H. In other words, we just defined a positive instance of conservative coalescing for $K = 0$ iff H is k-colorable. Furthermore, the initial graph G is greedy-2-colorable.

Notice that, if there is a coalescing f with at most K affinities not coalesced and such that G_f is k-colorable, there exists also a coalescing f' for which $G_{f'}$ is a k'-clique, with $k' \leq k$, thus a graph chordal and greedy-k-colorable. Indeed, to get f' from f, merge the vertices of G_f with the same color to get k vertices, then keep merging vertices not connected by an edge to get a k'-clique with $k' \leq k$. This proves that the problem is still NP-complete if we ask G_f to be not just k-colorable, but also greedy-k-colorable or k-chordal, two properties that a k-clique has. However, it is not NP-complete for a fixed K (number of non coalesced affinities) because, then, there is a polynomial number of solutions and the problem consists in repeatedly checking if the graph of a solution k-chordal or greedy-k-colorable, which is polynomial.

Previously, to obtain the k'-clique, we may have merged vertices not connected by affinities. To ensure that G_f can be obtained by merging only vertices connected by affinities, the proof must be modified as follows. A k-clique is added to G along with many affinities: one between each vertex of the clique and each vertex in V (but not with the x_e and y_e nodes). The instance of conservative coalescing built from $H = (V, E)$ is now an interference graph with $|V| + 2|E| + k$ vertices, $|E| + k(k-1)/2$ edges, and $2|E| + k|V|$ affinities. $k \leq |V|$—otherwise H would always be trivially k-colorable—thus the reduction is polynomial. For each of the vertices in V, at most one affinity among the k towards the clique can be merged. So, H is k-colorable iff there is a coalescing f

with at most (actually, exactly) $(k-1)|V|$ affinities not coalesced. Furthermore, in this case, G_f is a k-clique and can be obtained by merging only vertices connected by affinities. Therefore, the problem remains NP-complete even if one asks G_f to be greedy-k-colorable or chordal and one asks G_f to be obtained by merging only vertices connected by affinities.

The only remaining detail is that the interference graph G we used in the last reduction is not greedy-2-colorable anymore because it contains a k-clique. To complete the proof with all restrictions, each edge (u, v) of the clique is replaced by p edges (u_i, v) and p affinities $\langle u_i, u\rangle$, where $(u_i)_{1\leq i\leq p}$ are new vertices and $p > |V|$. As before, if H is k-colorable, there is a coalescing with at most $(k-1)|V|$ affinities not coalesced. Conversely, consider such a coalescing f. Suppose that two vertices u and v from the previous clique are merged by f, i.e., have the same color, then none of the corresponding $\langle u_i, u\rangle$ can be coalesced by f. Now, de-coalesce u from v, coalesce all p affinities $\langle u_i, u\rangle$, and give up coalescing the other affinities associated with u: these are the ones with nodes of V, thus at most $|V| < p$. By doing this for all pairs (u, v) that are merged in f, one gets a *strictly* better coalescing for which all vertices of the previous clique have a different color. Thus it has a cost greater or equal to $(k-1)|V|$, which is not possible because this the cost of f. Thus, in f, all affinities $\langle u_i, u\rangle$ are merged as well as all affinities $\langle x_e, y_e\rangle$ and H is k-colorable. □

In practice, conservative coalescing heuristics do not consider all affinities at the same time, instead, they consider them one by one, according to some priority—for instance, a higher priority is given to affinities corresponding to copies in nested loops. We call this strategy *incremental conservative coalescing*. Two incremental conservative tests exist, by Briggs et al. [1994] and George and Appel [1996], called respectively Briggs's and George's rules by Appel [1998].

Briggs Merging u and v is conservative if the resulting vertex has at most $(k-1)$ neighbors of degree at least k.

George Merging u and v is conservative if all neighbors of u with degree at least k are also neighbors of v.

These tests guarantee that the greedy-k-colorability property of the graph is maintained. Indeed, consider the elimination process that defines the greedy-k-colorability, i.e., the simplification scheme of Chaitin et al. [1981], described in Chapter 2 by Function `Is_kGreedy`, page 21. We recall that the principle is to remove nodes with $< k$ neighbors from the graph.

A vertex merged by Briggs's test can always be removed from the graph once its neighbors of degree $< k$ are removed, thus such a coalescing is always safe. The situation is slightly different for George's test: once the neighbors of degree $< k$ are removed, one ends up with a subgraph of the original graph, thus not harder to color. But if v cannot be removed from the original graph, the same is true for the merged vertex and the cost of spilling the two merged live-ranges is possibly larger. Thus, if George's test is used in a Chaitin-like allocator where spilling and coalescing are done in the same framework, the interaction with spilling is unclear. This is the reason why George's rule is used by George and Appel [1996] to merge a vertex u with v *only* if v is a *precolored* vertex (machine register), since these are never spilled.[3]

[3] To ensure this, in the original simplification scheme, they are not allowed to be simplified from the graph, and the algorithm stops whenever there remains only this k-clique.

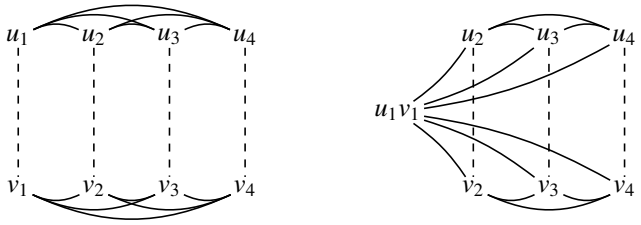


Figure 5.5: Local tests are not enough for coalescing: (a) permutation of size 4; (b) coalescing $\langle u_1, v_1 \rangle$ increases the degree to 6.

We point out however that, if spilling is done first, as done by Appel and George [2001] or Hack et al. [2006] for instance, to get a greedy-k-colorable interference graph, no spill will be done, hence George's rule can be used for *any* two vertices, resulting in more coalesced moves. The same applies for the last phase of Chaitin-like approaches, i.e., when no spill is introduced anymore.[4] We will see in Chapter 6 experiments that show this is indeed useful in practice.

When the register pressure is high, such local tests are not powerful enough, in particular if trying to coalesce parallel copies when Maxlive is close to the number of registers, as the experiments by Appel and George [2001] show. The problem is that the test is local and, even worse, it is done before the removal of "move-related" vertices of small degree from the graph. Figure 5.5a shows a permutation of 4 values. Assume $k = 6$. If affinities are coalesced one at a time and a local rule is used, the first merged vertex would become of degree 6 (Figure 5.5b); if its neighbors also have degree 6—due to other vertices not shown and not removed yet—a local rule will decide to not coalesce it.

Another deeper reason is due to the incremental nature of this form of coalescing. If G is a greedy-k-colorable graph and if S is a set of affinities that can be coalesced simultaneously to get a greedy-k-colorable graph, it may happen that coalescing any affinity in S leads to a graph that is not greedy-k-colorable. This is illustrated in Figure 5.6. The graph remains greedy-k-colorable if the two affinities $\langle a, b \rangle$ and $\langle a, c \rangle$ are coalesced, but not if only one is coalesced. To get a sequence of coalescing that is conservative at each step, one would need to consider affinities "obtained by transitivity" such as the pair $\langle b, c \rangle$ in Figure 5.6. This example shows that, *by essence*, a coalescing strategy that coalesces only one affinity at a time, conservatively, *cannot always reach the optimal conservative coalescing*.

One can try to improve these local conservative tests. As mentioned by George and Appel [1996], George's rule can be extended by considering that only the neighbors of u with at most $(k-1)$ neighbors of degree $\geq k$ need to be neighbors of v, but they state that "it would be more expensive to implement" hence did not use it. More generally, one can simply coalesce the move aggressively—i.e., merge the corresponding vertices if interferences allow it—and check, in linear time, whether the resulting graph is greedy-k-colorable or not. This is useful to get a much more efficient coalescing, as shown in Chapter 6. The same is true for a given set of moves. One can try to merge

[4] Although one does not know it is the last phase until the end of the phase. To exploit this, the last phase could be done twice.

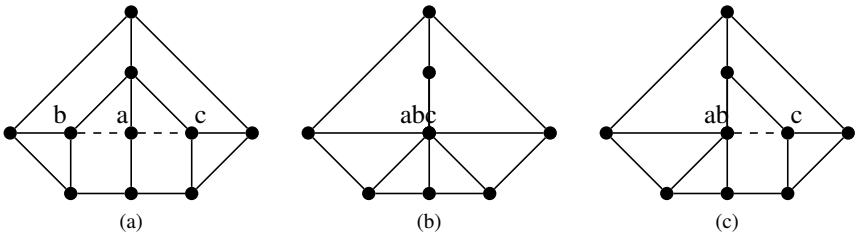


Figure 5.6: "Diamond" counter example for incremental coalescing: (a) remains greedy-3-colorable if both $\langle a, b\rangle$ and $\langle a, c\rangle$ are coalesced (b), but not just one (c).

all corresponding vertices and check if the graph is greedy-k-colorable. This lead to the idea of "incremental" coalescing, where one would like to know, for a given subset of affinities (possibly only one), if it is possible to coalesce all of them while staying k-colorable.

For instance, if G is k-colorable with a k-coloring f such that $f(x) = f(y)$, then there is of course a set of pairs of vertices, including the pair (x, y), that, once merged, lead to a greedy-k-colorable coalesced graph: simply merge all vertices with same color to get a k-clique. But, in general, one does not want to merge any two vertices, but to give more priority to vertices linked by affinities. In that case, which vertices should be merged? The dumb heuristic that coalesces all nodes colored equally does not answer this problem, neither do Briggs's and George's rules. This raises the question of the complexity of incremental conservative coalescing, which is the conservative coalescing problem for a *single* affinity.

Problem: INCREMENTAL CONSERVATIVE COALESCING
Instance. Graph $G = (V, E)$, one given affinity $a = \langle x, y\rangle$, an integer k.
Question. Can a be coalesced to get a k-colorable graph, i.e., is there a k-coloring f of G such that $f(x) = f(y)$?

Theorem 5.7 shows that this problem is NP-complete if G can be any k-colorable graph, i.e., knowing that G is k-colorable does not help to decide if it remains k-colorable after a single coalescing! However, as Theorem 5.8 states, the problem is polynomial if G is k-chordal. The complexity of the practical intermediate case, when G is greedy-k-colorable, would be of most interest, but is however still open.

Theorem 5.7. *Incremental conservative coalescing is NP-complete if G is an arbitrary k-colorable graph, even for $k = 3$.*

Proof. We use a reduction similar to the proof of Graph 3-colorability, i.e., with a reduction from 3SAT [Garey and Johnson, 1979, Problem LO1]. However, here, we will make a small detour through 4SAT. First, we show how to build, from an instance of 4SAT, a graph G that is 3-colorable iff there is an truth assignment for the 4SAT formula. Consider an instance of 4SAT, i.e., a set U of n variables $x_1, \ldots, x_n$, and a set C of m clauses $c_1, \ldots, c_m$, each with 4 literals $y_{i,1}, \ldots, y_{i,4}$. Each $y_{i,j}$ is a x_k or its negation. A graph $G = (V, E)$ is built as follows. It has three vertices T for true, F for false, and a third one, R, to form a triangle. For each variable $x_i \in U$, there are two vertices, denoted x_i and $\overline{x_i}$, which form a triangle with R. With 3 colors, this will force x_i and $\overline{x_i}$ to have the colors of T and F, or the converse. For each clause

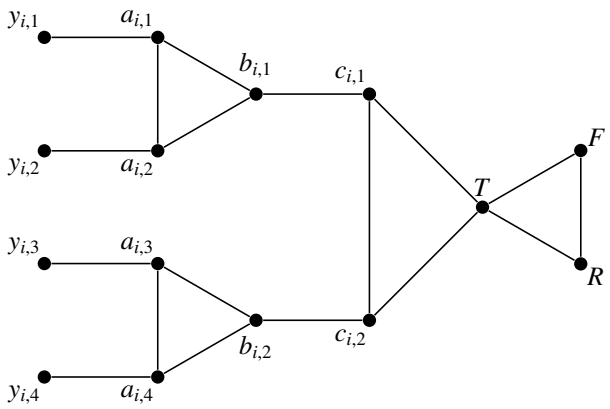


Figure 5.7: Reduction of incremental coalescing to 4-SAT.

$c_i \in C$, there are four vertices $a_{i,j}$, two vertices $b_{i,j}$, and two vertices $c_{i,j}$, connected as depicted in Figure 5.7. As for the original proof of graph k-coloring [Cormen et al., 1989, Page 962], it is easy to see that G is 3-colorable iff there is a truth assignment for the clauses. Indeed, if G is 3-colorable, then the four literals $y_{i,j}$ cannot be all colored as F, otherwise the two $b_{i,j}$ must be colored as F, and one of the two $c_{i,j}$ cannot be colored. Thus interpreting the colors of each x_i gives a truth assignment. Conversely, if there is a truth assignment, color each x_i as T iff it is true in the 4SAT formula. Then, color $b_{i,1}$ as T (resp. F) if $y_{i,1}$ or $y_{i,2}$ is true (resp. both are false), the same for $b_{i,2}$. The rest of the 3-coloring follows.

Now consider an instance (U, C) of 3SAT. Add a new variable x_0 and define an instance (U', C') of 4SAT, where $U' = U \cup \{x_0\}$ and each clause $c'_i \in C'$ is defined from $c_i \in C$, by $c'_i = y_{i,1} \vee y_{i,2} \vee y_{i,3} \vee x_0$ if $c_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$. Notice that there is a truth assignment for C' by simply setting x_0 to true. Moreover, there is a truth assignment for C iff there is one for C' for which x_0 is false. Finally, define a graph G from C' as before and consider the affinity $\langle x_0, F \rangle$. From the previous study, G is 3-colorable, by coloring x_0 as T. Furthermore, there is a 3-coloring of G such that the vertices x_0 and F have the same color—i.e., are coalesced—iff there is truth assignment for C' for which x_0 is false. □

This theorem is not encouraging, as it proves we cannot guarantee the optimality of a single step of an incremental heuristic which coalesces affinities one by one. Hopefully, there follows a less depressing result whenever the initial graph G is chordal.

Theorem 5.8. *Incremental conservative coalescing can be solved in polynomial time if G is chordal.*

Proof. Let $G = (V, E)$ be a chordal graph and $\langle x, y \rangle$ be the affinity to coalesce. A fundamental property [Golumbic, 1980, Thm. 4.8] is that G can be represented as the intersection graph of a family of subtrees $(T_v)_{v \in V}$ of a tree T. We use the word nodes for the vertices of T to distinguish them from the vertices of G. The nodes of T are the maximal cliques of G (for inclusion), each vertex $v \in V$ corresponds to a subtree T_v, and $(u, v) \in E$ iff T_u and T_v intersect. A chordal graph with the tree representation can be easily colored with any $k \geq \omega(G)$ colors, starting from any node n of T. Orient the tree T to get a directed tree with root n and color the subtrees that contain n. Then, go

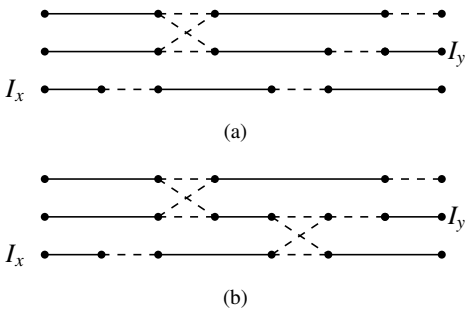


Figure 5.8: Incremental coalescing for chordal graphs, using covering by intervals: (a) I_x and I_y cannot have the same color; (b) they can.

down the branches of the tree and, at each new node, color the subtrees that start at this node with the available colors. This coloring is always possible because, at each node, at most $\omega(G)$ subtrees intersect. Furthermore, there is no cycle in T so no coloring decision can lead to a conflict.

Now the question is: Is it possible to color G with k colors so that x and y have the same color? This question can be answered in polynomial time as follows. We assume that T_x and T_y do not intersect and $k \geq \omega(G)$, otherwise the answer is trivially "no." Let P be the shortest path on T between T_x and T_y. It starts at a node n_x of T_x, and ends at a node n_y of T_y, and none of the intermediate nodes of P are in T_x or T_y. The intersection of the subtrees $(T_v)_{v \in V}$ with P are *intervals* $(I_v)_{v \in V}$. We add new short "dummy" intervals, containing a single node, so that all nodes of P are contained in exactly $\omega(G)$ intervals. We claim that T_x and T_y can have the same color iff there is a set of disjoint intervals, including I_x and I_y, that covers all nodes in P. Indeed, if G has a k-coloring such that x and y have the same color, then the intervals with the same color than x and y, in addition to some dummy intervals, provide such a covering. Conversely, if such intervals exist, one can merge all the subtrees with the same color as x and y, including the dummy intervals, to get the representation of a new chordal graph G' with $\omega(G') = \omega(G) \leq k$; it can thus be colored with k colors and this coloring corresponds in G to a k-coloring where x and y have the same color.

It remains to show how to find such a set of intervals in polynomial time. This can be done as follows: represent the intervals horizontally on $\omega(G)$ lines, all full because of the dummy intervals added. There is a cover of P with disjoint intervals, including I_x and I_y, iff there is a path from the line of I_x to the line of I_y, following intervals and possibly changing line only from the end of an interval to the beginning of another (i.e., contiguous intervals). This can be checked in $O(V\omega(G)) = O(V^2)$ by a simple marking process from left to right. See Figure 5.8 for an illustration where dotted lines represent the possible changes of line. □

Theorem 5.8 shows that one could design an incremental conservative coalescing strategy for chordal graphs. If G is chordal and $\langle x, y \rangle$ is an affinity that one absolutely wants to coalesce because the corresponding move is expensive, it can be decided if this is possible. But then, if the affinity is coalesced, the graph may not be chordal anymore. However, it is still possible to make it chordal by an appropriate merge of vertices as done in the proof of the theorem. But, these merges will artificially create long live-ranges (subtrees) and increase locally the register pressure if merging with dummy intervals. This may forbid the coalescing of more important affinities afterwards. A

better strategy would be to stay in the class of greedy-k-colorable graphs that is larger than the class of chordal graphs. Unfortunately, we do not know the complexity of this problem yet. However we will see in Chapter 6 a heuristic for this problem that uses the algorithm we just described.

5.4 Complexity of optimistic coalescing

If G is greedy-k-colorable, coalescing as many moves as possible so that the coalesced graph is k-colorable, or even greedy-k-colorable, is NP-complete as stated by Theorem 5.6. To approximate this problem, incremental conservative coalescing coalesces moves one by one, so that the graph remains greedy-k-colorable but, of course, with no guarantee that the chosen moves are the right ones. Even worse, as shown in Section 5.3, it may happen that no such conservative sequence exists. Park and Moon [1998, 2004] proposed a "dual" approach, *optimistic coalescing*. A first phase of aggressive coalescing coalesces moves regardless of the k-colorability of the graph. Then, a second phase gives up on some moves, i.e., "de-coalesces" them so that the graph becomes greedy-k-colorable again.

If most moves can be coalesced, this approach can be more effective than using a too-conservative local test such as the tests of Briggs or George. However, in practice, it is not clear which moves should be coalesced aggressively in the first phase: remember that, by Theorem 5.5, aggressive coalescing is NP-complete too. Moreover, even if all moves can be aggressively coalesced, it is not clear which moves should be de-coalesced in the second phase. The goal of this section is to address the complexity of this second problem. If one requires the de-coalesced graph to be just k-colorable, it is of course NP-complete as the first part of the proof of Theorem 5.6 shows: after all affinities are coalesced, it is hard to decide if the resulting graph is k-colorable or not, i.e., if some de-coalescing needs to be done. In practice however, the graph should be more than just k-colorable, it should be easy to color, for example greedy-k-colorable. So, the interesting instance of optimistic coalescing can be formulated as follows.

Problem: OPTIMISTIC COALESCING
Instance. Graph $G = (V, E)$ greedy-k-colorable, affinities $\mathcal{A}$ that can all be coalesced aggressively (i.e., there is a coalescing f of G such that $\forall \langle u, v \rangle \in \mathcal{A}$, $f(u) = f(v)$), integers k and K.
Question. Is there a de-coalescing of G_f (i.e., a coalescing g of G such that $g(u) = g(v)$ implies $f(u) = f(v)$), such that at most K affinities $\langle u, v \rangle$ are not coalesced (i.e., satisfy $g(u) \neq g(v)$) and such that G_g is greedy-k-colorable?

Theorem 5.9. *The optimistic coalescing problem is NP-complete, even for $k = 4$, and even if G is chordal.*

Proof. The proof is by reduction from Vertex Cover [Garey and Johnson, 1979, Problem GT1], which is NP-complete even if all vertices have degree at most three [Garey et al., 1976]. Let $H = (V, E)$ be a graph such that all vertices have degree at most 3. The instance of optimistic coalescing is built as follows. For each node $v \in V$, there is a structure as shown on Figure 5.9. Each of the three "hexagons" in this structure is a widget as shown on the right part of the figure. The central vertex c_v is in fact two vertices c_v and c'_v linked by an affinity. c_v belongs to the inner 4-clique, while c'_v has neighbours v_1, v_2 and v_3, hence they are both of degree 3. On this structure, each of the

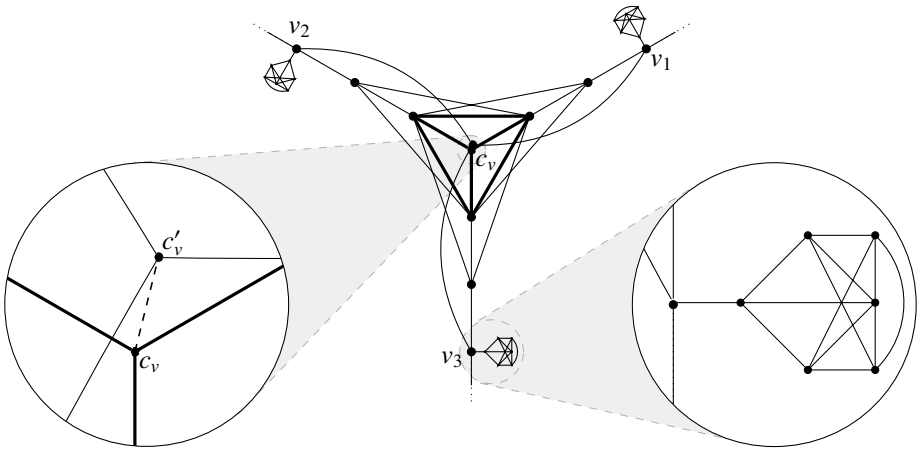


Figure 5.9: Reduction for optimistic coalescing: vertex structure and ad-hoc widget. c'_v is connected to v_1, v_2 and v_3 but c_v is not.

three vertices $(v_i)_{1\leq i\leq 3}$, can be used to connect v to one of its neighbors. Since v has at most three neighbors, the whole graph H can be transformed into this format, creating a new graph G. G is not chordal, but we will show later how to make it chordal. It can be completely coalesced, forming G_f. Our first goal is to de-coalesce some of the pairs $\langle c_v, c'_v\rangle$ so as to get a graph G_g that is greedy-4-colorable.

The important point is to understand how the greedy-4-coloring algorithm can "eat" a structure. It can only works if there is at least one node of degree < 4. All the vertices of the hexagonal widgets have degree ≥ 4 so the structure cannot be eaten from these. If the structure for $v \in V$ has no neighbor, either because v has no neighbor in H or because the neighbor structures have already been eaten, then each v_i has degree 3: they can be eaten, then the hexagonal widgets and the inner structure can be eliminated too. Finally, notice that the structure cannot be completely eaten from just two of its branches: even if only one of the v_i remains, the inner 4-clique—represented in bold—cannot be removed. Hence the only remaining possibility is to attack the structure from c_v and c'_v, which is possible only if they are not coalesced. This shows that there are only two ways for the greedy algorithm to eat the structure corresponding to a vertex v of H: either after having eaten all the structures corresponding to the neighbors of v, or by de-coalescing c_v and c'_v and attacking the structure from the heart.

The previous study shows that G_f after de-coalescing is greedy-4-colorable iff, for each $(u, v) \in E$, a de-coalescing occurred in at least one of the structures corresponding to u and v, i.e., iff the set of vertices u such that a de-coalescing occurred in the corresponding structure in G is a vertex cover for H. Hence, we have proved that de-coalescing to obtain a greedy-k-colorable graph is NP-complete.

Finally, for what we want to prove, G is not enough. We need to build a greedy-4-colorable graph G' (even chordal if possible) and affinities such that all affinities can be aggressively coalesced into G_f and such that these new affinities will not be chosen to de-coalesce optimally G_f into a greedy-4-colorable graph. In G, there are three kinds of chordless cycles: in the hexagonal widgets, inside each structure because there is a chordless cycle including (c'_v, v_i, v_j), and between structures if H itself is not chordal.

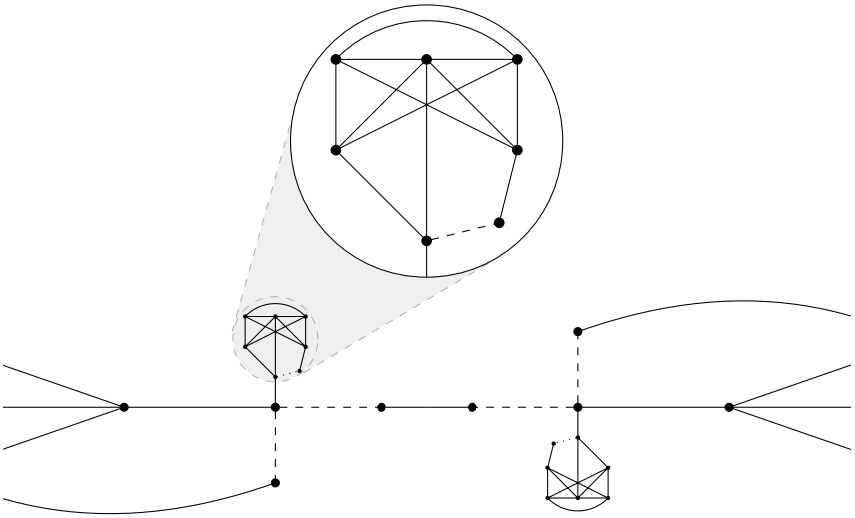


Figure 5.10: Optimistic reduction: adding affinities to obtain a chordal graph.

These cycles are broken by introducing some affinities as shown on Figure 5.10. The reduction is still correct because it is always better to choose to de-coalesce an affinity $\langle c_v, c'_v \rangle$ instead of any other affinity in the structure: this allows to eat the whole structure with a single de-coalescing.

To conclude, G' is chordal, greedy-4-colorable, and all affinities can be aggressively coalesced. Furthermore, one can de-coalesce at most K affinities to get a greedy-4-colorable graph iff H has a vertex cover of size at most K. Using Property 5.4, this proves that optimistic coalescing is NP-complete for any fixed $k \geq 4$.

□

5.5 Summary and conclusion

Our complexity study addresses all variants of register coalescing introduced in the literature: aggressive coalescing, conservative coalescing, incremental conservative coalescing, and de-coalescing. Due to the spilling phase and coloring mechanism, the coalescing phase may have to deal with particular graphs only, for example k-colorable after enough spill, greedy-k-colorable for Chaitin-like coloring, or k-chordal for SSA-like splitting. The goal was to check whether, when restricted to such graphs, coalescing remains hard or if some polynomial instances exist. Such a complexity study has never been done before. We now summarize our results, discussing the link between the different coalescing variants.

The aggressive coalescing problem is to remove as many moves as possible regardless of the colorability of the resulting graph. This optimization, used by Sreedhar et al. [1999]; Budimlić et al. [2002], and Rastello et al. [2004], arises for instance when translating out of SSA, independently of register allocation, in an earlier phase. It is also the first phase of optimistic coalescing, which first coalesces in a non-conservative way. Rastello et al. [2005] proved that, in the context of SSA, this problem is NP-complete when the size of the largest ϕ-function is unbounded. For completeness, we

refined this result: Theorem 5.5 shows that it is NP-complete even if ϕ-functions have a fixed size—at least two arguments—and the program contains only three interferences. It thus shows that coalescing is hard, even without any constraints on the number of registers.

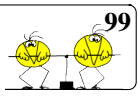
The conservative coalescing problem is to remove as many moves as possible while keeping the graph k-colorable. While the motivation of aggressive coalescing is to let the register allocator find adequate split points later, by some de-coalescing, the idea of conservative coalescing is to consider initial split points as good points for coloring. To prove the NP-completeness of conservative coalescing, we could have used our reduction for aggressive coalescing (Theorem 5.5): in this particular case, requiring that the coalesced graph is k-colorable is actually not a constraint since it is a 3-clique. Another reduction is given by Hack et al. [2005]. We preferred to refine the reduction mentioned by Appel and George [2001], maybe more natural because directly related to graph coloring. We showed it is NP-complete even if the initial graph is greedy-2-colorable and one requires the coalesced graph to be chordal or greedy-3-colorable.

The incremental conservative coalescing problem corresponds to a pragmatic approach to address the conservative coalescing problem. The idea is to incrementally coalesce variables, one by one, for example choosing the more expensive first, while keeping the graph k-colorable. This approach corresponds to Chaitin-like register allocation heuristics for which the conservative tests used by Briggs [1992] and George and Appel [1996] are not exact. Testing if a given coalescing maintains the greedy-k-colorable (resp. k-chordal) property of the resulting graph is of course polynomial. If the answer is in the affirmative, then the process can continue; if not, the natural questions behind this study are: "is the resulting graph still k-colorable?" and "would more coalescing make it greedy-k-colorable (resp. k-chordal) again?" The pessimistic result given by Theorem 5.7 is that this problem is still NP-complete if the initial graph is k-colorable. Even coalescing a single move is hard for an arbitrary graph. On the contrary, Theorem 5.8 provides a polynomial solution if the initial graph is chordal. The similar problem for a greedy-k-colorable graph is still open. But the result on chordal graphs provides openings to the design of new heuristic solutions: from a k-colorable graph obtained after a single coalescing, we are able to slightly modify the program to obtain a k-chordal graph again, or a greedy-k-colorable graph. We did derive a heuristic from this result which we will explain in Chapter 6.

The last problem addressed in this paper, the optimistic coalescing problem, is about coalescing aggressively, then giving up as few moves as possible so that the resulting graph becomes k-colorable again. This dual approach proposed by Park and Moon [2004] provides a more efficient heuristic in practice than the classical conservative approach, as shown by Appel and George [2001]. However we will see in Chapter 6 that the effectiveness of optimistic coalescing over conservative coalescing is not to be taken for sure, and that more involved conservative strategies can outperform optimistic ones. The conservative approach coalesces from a non-coalesced graph as long as the colorability test is satisfied whereas the optimistic approach de-coalesces from an aggressively coalesced graph as long as the colorability test is unsatisfied. Our initial intuition was that, if for a given graph, greedy-k-colorable or k-chordal, there exists a *set* of coalescings such that the resulting graph has the same property, then there exists a *sequence* of single coalescings such that each intermediate graph has the same property. Unfortunately, this intuition is false as illustrated by Figure 5.6 and this could give good reasons to think that, in practice, an optimistic approach will behave better than a conservative approach. But our experiments in Chapter 6 show that a good conservative coalescing provides better results, apparently because the aggressive part of optimistic

coalescing can make some "bad choices"—good for the aggressive, but bad for the conservative because hard to undo. Since aggressive coalescing is NP-complete, the interesting statement for the optimistic coalescing problem is then: given an aggressively coalesced graph, how to de-coalesce a minimum number of moves such that the resulting graph becomes greedy-k-colorable (resp. k-chordal) again? Theorem 5.9 shows that this problem is NP-complete even for $k = 4$.

To conclude, our study shows that most variants of the coalescing problem are NP-complete—which is certainly not a surprise but was never really proven before formally and in such details—and confirms the practical importance of chordal and greedy-colorable graphs. We believe that their properties were maybe not yet completely exploited for the design of good conservative coalescing heuristics and good de-coalescing heuristics, so we developed new heuristics based on these properties. There was indeed space for improvements, which we will present in the next chapter.



Maier's Law:
If the facts do not conform to the theory, they must be disposed of.
Corollaries:

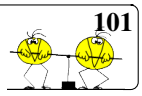
1. *The bigger the theory, the better.*
2. *The experiment may be considered a success if no more than 50% of the observed measurements must be discarded to obtain a correspondence with the theory.*

6 Advanced coalescing: improving the coloring

This chapter is the experimental counterpart of the previous chapter. While Chapter 5 studied the complexity of the coalescing problems, this chapter is devoted to the study of practical coalescing solutions. As such, we will not re-introduce the whole coalescing problem again, but will instead focus on the important points from an experimental point of view.

Coalescing plays an important role during register allocation, since it allows to diminish the number of register-to-register moves executed by a program. Since the first graph-based register allocator by Chaitin et al. [1981], coalescing has been part of the register allocation optimizations. Where do the register copies come from in a program? High-level source codes contain few copy (`move`) instructions, and a pass of copy folding under Static Single Assignment (SSA) can easily remove the unnecessary ones. But many optimizations phases in a compiler insert copies rather than more aggressively rewriting the code. Since Chaitin et al., the assumption has been that the moves can be removed relatively easily. For example, Cooper et al. [2001] leave scads of copies, relying on a Chaitin-Briggs allocator to perform copy coalescing. The SSA form also introduces virtual "ϕ-functions" at join points of the control-flow graph (CFG), which semantically correspond to parallel copy instructions placed in blocks on the incoming control-flow edges. When going out of SSA, these copies must be either carefully removed [Sreedhar et al., 1999; Rastello et al., 2004], or eliminated by a later register allocation phase. Copies between registers can also be added to handle, in a simple way, register constraints or calling conventions (see Chapter 8, Section 8.1.1.2). A later phase is supposed to remove these additional copies. Our final example is live-range splitting, which is used to improve spilling (see Chapter 2, Section 2.2.3, and the discussions in Chapter 3).

In the context of register allocation in two phases—which we advocate since Chapter 3—, the first "spilling" phase should be free to use as many splits as required to get the best solution in terms of costs of the `load` and `store` instructions added. This is only effective if the second "coalescing" phase is powerful enough to remove most of the unnecessary copies. The extreme situation is when live-ranges are split at *each* program point so as to formulate the spilling problem as an integer linear programming (ILP) problem, as proposed by Appel and George [2001]. Loads and stores are then nicely optimized but many copies are created that need to be removed. This shows there are many reasons to try to get rid of copy instructions at the assembly-code level. This is the goal of register coalescing, which does this during register al-



location and thus is tightly connected to spill optimization and register assignment. However, we have seen in Chapter 5 that most problem related to coalescing are NP-complete, except for one particular case (incremental coalescing on a chordal graph), hence heuristics are used.

As explained in the previous chapter, the initial proposal of Chaitin et al. [1981] was to coalesce moves, before the simplify phase, in an *aggressive* fashion, i.e., regardless of the colorability of the resulting graph. The effect is that many moves are indeed deleted but the number of potential and actual spills almost always increases. Chaitin et al. worked on a machine where loads, stores, and moves had the same cost, so their approach was justified. Briggs [1992] explored what he called "aggressive live-range splitting" in his thesis, which he uses before attempting to color the interference graph, so that the allocator does not spill entire live-ranges. Obviously, he could not use the same aggressive coalescing as Chaitin et al. since it would cancel the splitting. So, he introduced *conservative coalescing*, which consists in coalescing a move only when one can ensure that, if the initial graph was R-colorable, the resulting graph is still R-colorable. In Briggs's test, a move is coalesced only if the resulting node has at most $(R-1)$ neighbors with of high degree, i.e., $\geq R$. George and Appel [1996] then proposed Iterated Register Coalescing (IRC), a fully-conservative approach where conservative coalescing is intermixed with the simplify phase and no aggressive coalescing is performed.[1] We described the general mechanism of IRC in Chapter 2, Section 2.2.4. As the coalescing test in IRC is fast but not exact (the test can fail even if the move can be conservatively coalesced), better results are obtained if moves are tested several times while simplifying nodes. For that, worklists of potentially coalescable moves are created and updated during the simplify phase. This increases the running time of the complete register allocator, even if the smart implementation strategies defined by Leung and George [1999b] can reduce this overhead.

IRC is a very popular graph-based register allocator, mainly because it manages to stay simple in its design while still giving a correct quality of resulting code. However, in terms of moves optimization, its authors where not satisfied when they investigate their ILP based "optimal spill" [Appel and George, 2001]. Indeed, we already told they needed to perform an "extreme" aggressive splitting, which split variables at every program point. Faced with this intense need for coalescing, the IRC performed poorly and Appel and George [2000] launched in August 2000 the "Optimal Coalescing Challenge," a database of 474 graphs obtained after a phase of extreme splitting followed by optimal spilling. The goal was to find the best coalescing solutions for all graphs, or, at least, to come up with better solutions than the ones obtained with their IRC.

By December 2000, they already obtained satisfying results. They found a heuristic based on work by Park and Moon [1998, 2004], who had proposed an allocator in which the coalescing was "optimistic." *Optimistic coalescing* relies on the fact that, although coalescing can increase the degree of the merged nodes, it can also decrease the degree of common neighbors, which is a positive effect. Thus, it seems better to perform aggressive coalescing first and then to decide, during the select phase, to undo some coalescings to avoid spilling. In the Park and Moon algorithm, when a coalesced node is to be spilled, it is split back (de-coalesced) into separate nodes. Some of them are colored with a common color if possible, the others are either colored at the very end of the select phase or spilled if no colors are available for them. Appel and George adapted the optimistic heuristic of Park and Moon because, in their case, spilling was

[1] But for an optional constant propagation phase under SSA, performed beforehand and not included in the scheme.

already decided. Despite its naïve de-coalescing phase, the optimistic coalescing takes benefit from the aggressive phase and outperforms IRC.

In this thesis, we advocate that register allocation should be performed in two phases, following recent work from Appel and George [2001]; Brisk et al. [2005] and Hack and Goos [2006]: a first spilling phase, with live-range splitting, that inserts loads, stores, and moves so that the resulting interference graph is greedy-R-colorable; and a second phase that coalesces moves and assigns colors to variables. In this context, register coalescing is crucial to reduce the cost of moves added blindly by live-range splitting. Unlike classical graph-based approaches, it is a pure coalescing problem, with no additional spill: how to coalesce moves in a greedy-R-colorable graph so that it remains R-colorable? The same problem arises in the last iteration of Chaitin-like register allocators, i.e., when no potential spill is needed. This pure coalescing phase looks the same as in the context of the Coalescing Challenge, for which Appel and George [2000] adapted the optimistic coalescing of Park and Moon [2004]. But there are reasons to revisit this problem: First, we believe the results can be improved; Second, the live-range splitting of Appel and George is extreme, and the graphs obtained are very particular: all nodes in the interference graph have degree at most $(R-1)$, but for at most R nodes that represent the machine registers and form an R-clique. Hence, their approach does not work for general greedy-R-colorable graphs. In this chapter, our goal is to address the following questions:

- Can we take advantage of the fact that the initial graph is greedy-R-colorable, i.e., that no spill more spill is needed, to improve coalescing?
- Can we derive more involved conservative tests?
- Can we avoid testing each copy several times as in IRC?
- Can we adapt Park and Moon optimistic coalescing to greedy-R-colorable graphs and improve the de-coalescing phase?
- Is incremental conservative coalescing really worse than optimistic coalescing? (i.e., is it really worse to coalesce moves one by one, while keeping the graph R-colorable, than to coalesce moves aggressively, then possibly de-coalesce to get an R-colorable graph?)

We developed advanced conservative and optimistic coalescing algorithms that allow us to give affirmative answers to the first four questions. Then, the evaluation of these new strategies let us think that the answer to the last question might well be *no*, for greedy-R-colorable graphs. Section 6.1 recalls some necessary definitions and elementary properties linked to register coalescing. Section 6.2 presents our more involved conservative tests to decide if the graph remains R-colorable after a given move is coalesced. These tests are used in an incremental approach whose results outperform, by roughly 15%, state-of-the-art optimistic algorithms, even though it is conservative! In Section 6.3, we improve optimistic coalescing by developing advanced de-coalescing mechanisms. Section 6.4 presents optimal coalescing rules that can be used to reduce by about 75% the size of the graphs, and improve the results as well. Section 6.5 is an analysis of our results on the collection of interference graphs provided by Appel and George [2000], the "Coalescing Challenge," which is now considered as the benchmark suite for evaluating coalescing algorithms. We evaluate variants and trade-offs between running times and quality of results, comparing also with optimal solutions provided by

Grund and Hack [2007] (found using ILP). Section 6.6 concludes, discussing possible improvements and open problems.

6.1 Recalling the coalescing problems

We will first recall the background information that will be extensively used in this chapter. These definitions can also be found in Chapter 2 but we prefer to repeat them here for the sake of clarity.

The *interference graph* $G = (V, E)$ is an undirected graph where each node $v \in V$ corresponds to a live-range of the program. There is an *interference* $e = (u, v) \in E$ if and only if (iff) u and v cannot share the same register. In addition to interferences, each copy instruction, also called *move*, is represented by an *affinity* $a = \langle u, v \rangle$. In general, an affinity has a weight $w(a)$ that gives an evaluation of how often the corresponding copy instruction would be executed. A *coloring* of G is a function $f : V \to \mathbb{N}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$. When $f(V)$ contains k different values, it is a k-coloring. Given a set of affinities $\mathcal{A}$, a *coalescing* is defined by a coloring f with no constraint on the number of colors. An affinity $a = \langle u, v \rangle \in \mathcal{A}$ is coalesced if $f(u) = f(v)$. The *coalesced graph* G_f is obtained from G by merging any two nodes linked by a coalesced affinity.

Section 2.2.2.5 defines as *greedy-k-colorable* a graph such that removing successively all nodes of degree $< k$ leads to the empty graph, i.e., if Function `Is_kGreedy`(G) returns TRUE. The pseudo-code of this function was given in Chapter 2, page 21, but we prefer to include it also here for completeness.

Function `Is_kGreedy`(G)

Data: Undirected graph $G = (V, E)$; $\forall v \in V$, degree[v] = #neighbors of v in G, k number of colors

```
1 stack = ∅ ; worklist = {v ∈ V | degree[v] < k} ;
2 while worklist ≠ ∅ do
3     let v ∈ worklist ;
4     foreach w neighbor of v do
5         degree[w] ← degree[w]-1 ;
6         if degree[w] = k − 1 then worklist ← worklist ∪ {w}
7     push v on stack ; worklist ← worklist \ {v} ;        /* Remove v from G */
8 if V = ∅ then return TRUE else return FALSE
```

After the graph is emptied, nodes can be popped from the stack and colored if needed, picking for each node a color not used by its $< k$ already-colored neighbors. Because under SSA the interference graph of a program is chordal, chordal graphs will have their importance there. Remember that Property 2.23 states that *a k-colorable chordal graph is greedy-k-colorable*. In other words, the simplify/select phases of graph coloring register allocators always succeed to color a chordal graph with k colors if it is k-colorable.

In the next sections we will consider several coalescing problems that arise in the heuristics used in register coalescing or register allocation. The complexity of these problems has been studied previously in Chapter 5; we recall here their formulation in a more informal way, with links to their corresponding formal definitions from the previous chapter. We recall that coalescing problems can be unweighted—optimization

of the number of static moves—or weighted—optimization biased by an approximate dynamic execution count.

Aggressive coalescing (page 87) consists in finding a coloring f (with no restriction on the number of colors) such that the cost of affinities not coalesced is minimized, i.e.,

$$\text{minimize} \sum_{a \in U} w(a) \quad \text{with} \quad U = \{\langle u, v\rangle \in \mathcal{A} \mid f(u) \neq f(v)\}$$

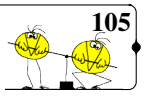
A simple heuristic for this NP-complete problem is to sort affinities by decreasing weights and to coalesce each affinity, one after the other, if no interference prevents it. Some heuristics for out-of-SSA conversion [Sreedhar et al., 1999; Rastello et al., 2004] try to exploit the structure of ϕ-functions and consider simultaneously several affinities corresponding to the moves of one ϕ-function, but they have never been integrated into a unified "coloring-coalescing" scheme.

Conservative coalescing (page 88) consists in finding a k-coloring f such that the cost of affinities not coalesced is minimized. It is NP-complete even if one asks the graph G_f to be greedy-k-colorable. A traditional heuristic is to consider affinities, one after the other, so that, after each coalescing, the graph remains k-colorable. (We are not aware of any heuristic that can consider several affinities simultaneously.) Such approaches are called *incremental*.

Incremental conservative coalescing (page 92) considers affinities one after the other. In such an approach, one has to answer, for each considered affinity $a = \langle u, v\rangle$, the following question: is there a k-coloring f such that $f(u) = f(v)$? This problem is NP-complete for a general graph and polynomially solvable for a chordal graph. We indeed gave a conceptual algorithm for this result in the proof of Theorem 5.8. Here, in Section 6.2, we give a linear-time algorithm, which can be used as a heuristic for greedy-k-colorable graphs—as for these graphs, the problem complexity remains open. Note however that asking G_f to be not only k-colorable but also greedy-k-colorable is easy since this property can be checked in polynomial time using Function `Is_kGreedy` after coalescing a. We study this *brute-force coalescing* in Section 6.2.1.

De-coalescing is linked to *optimistic coalescing* (page 95): how to undo an aggressive coalescing f to go back to a k-colorable graph. In other words, *de-coalescing* consists in finding a k-coloring g, where $g(u) = g(v)$ implies $f(u) = f(v)$, that minimizes the cost of affinities not coalesced. It is NP-complete even if the aggressive phase succeeded to coalesce all affinities; the difference with the optimistic problem of Chapter 5 is that we do not consider here that the aggressive phase can coalesce all affinities as in practice this is often not true. Also, one usually seeks a de-coalescing g such that G_g is not only k-colorable but also greedy-k-colorable.

In the next sections, we will explain existing coalescing heuristics, develop advanced heuristics and compare them to the first ones.



6.2 Conservative coalescing

We are aware of two existing conservative tests: the Briggs's rule by Briggs et al. [1994] and George's rule by George and Appel [1996]. Let us examine why they are conservative when applied on a greedy-k-colorable graph.

Briggs merges u and v if the resulting node has less than k neighbors of high degree, i.e., $\geq k$. This node can always be simplified after its neighbors of degree $< k$ are simplified, thus the graph remains greedy-k-colorable.

George merges u and v if all neighbors of u with high degree are also neighbors of v. After coalescing and once all neighbors of degree less that k are simplified, one gets a subgraph of the original graph, thus greedy-k-colorable too.

Originally, these rules were used for any graph, not necessarily greedy-k-colorable, and with an additional clique of pre-colored nodes—the physical machine registers. In this context, the rules have two restrictions. Since pre-colored nodes should never be spilled, they are never simplified, hence Briggs's rule does not apply to pre-colored nodes. And George's rule applies only if v is pre-colored, otherwise, if the graph is not greedy-k-colorable, there is a risk of spilling u and v instead of only u; this cannot happen with pre-colored nodes since they are never spilled. We first make a simple but important remark: *for greedy-k-colorable graphs, both rules can be used for any two nodes*. For George's rule, this is obvious as there is no spill for a greedy-k-colorable graph. For Briggs's rule, we can also decide to simplify pre-colored nodes, when possible, as any other node. Indeed, they form a clique and will thus be given different colors in any coloring. To get back to the original colors of these *pre*-colored nodes, a simple permutation of colors does the trick. Hence Briggs's rule also applies to pre-colored nodes.

Surprisingly, extending Briggs's and George's rules to any two nodes already leads to significant improvements (see Section 6.5). However, they still give insufficient results to coalesce the many moves introduced, for example, by a basic out-of-SSA conversion. The reasons are twofold. First, both rules are local decisions: they depend on the degree of neighbors only. But these neighbors may have a high degree just because their neighbors are not simplified yet, i.e., the test may be applied too early in the simplify phase. This is the reason why George and Appel [1996] proposed the *Iterated* Register Coalescing (IRC): instead of giving up coalescing when the test fails, the affinity is placed in a sleeping list and "awakened" when the degree of one of the nodes implied in the rule changes. Thus, affinities are in general tested several times, and move-related nodes—nodes linked by affinities with other nodes—should not be simplified too early to ensure the affinities get tested. The second reason is that these two tests are used to coalesce affinities in a sequential way, requiring that the graph stays greedy-k-colorable at each step. This is a limitation, as we explained in Chapter 5, Figure 5.6. In the following sections, we will try to overcome these two limitations.

6.2.1 Brute-force conservative coalescing

To address the limitation that current coalescing rules are too "local," we developed a more expensive test based on the fact that greedy-k-colorability is easy to check. The goal was to have a starting "brute-force" algorithm that could give an idea of how far the existing rules are from the "best" conservative rule. As in any incremental

approach, we consider affinities one by one in decreasing order of weights. To test an affinity, instead of using an overly-conservative local rule, the two corresponding nodes are merged aggressively, then, ignoring the other affinities, a complete simplify phase is done—using Function `Is_kGreedy`, page 104—to test if the resulting graph is greedy-k-colorable. If this brute-force test fails, the coalescing is not conservative and the two nodes are de-coalesced, i.e., kept separate as they were. The pseudo-code for this test is Function `Brute_Test`.

Function `Brute_Test`(G, u, v)

Data: Graph $G = (V, E)$, nodes u and v
Output: TRUE if u and v can be conservatively coalesced, FALSE otherwise.
Result: u and v are merged in G if it stays greedy-k-colorable.

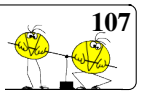
```
1 merge u and v into uv in G ;
2 if Is_kGreedy(G) = FALSE then
3 |   un-merge u and v in G ;
4 |   return FALSE
5 else return TRUE
```

We modified IRC so that, instead of using Briggs's and George's rules for conservative coalescing, it used Function `Brute_Test`. And it was clear that this test is much more powerful than these local rules. This means that, if the classic rules of Briggs and George are certainly not good enough, it is *not* because conservative rules are intrinsically bad, but because these ones are too local. Hence it seems possible to devise better conservative rules. Of course, the `Brute_Test` comes to mind, but it is more costly than a local rule such as Briggs's and George's tests, as its complexity is linear in the graph size. Thus the overall complexity for testing each affinity once would be $O(|\mathcal{A}|\,(|E| + |V|))$.

IRC may seem to be a linear algorithm. However, as mentioned earlier, there is an overhead due to the fact that affinities are evaluated several times. The algorithm gives up coalescing only when the worklist of affinities is empty: a node is chosen and its affinities are removed. It can then be simplified and the process continues with the remaining nodes and affinities. Leung and George [1999b] propose a counting mechanism to reduce the number of useless evaluations but, still, the overall complexity is *not* linear. Here, when `Brute_Test` returns FALSE, we have two choices, either keep the affinity in some sleeping list for a possible re-evaluation, as in IRC, or immediately give up coalescing this affinity. We observed that, because `Brute_Test` is more powerful, testing each affinity only once degrades only marginally the quality of the result. On the contrary, keeping affinities and reconsidering them each time the graph changes is far too costly. In other words, we get an acceptable trade-off between execution time and performance by spending more time in the test but avoiding the re-evaluation of affinities.

With little effort, we improved the idea of the `Brute_Test` into a full coalescing algorithm, whose pseudo-code is given in Function `Brute_Force_Improved`, page 108. Its functioning resembles the one of IRC, but without the need for a spilling process, nor for a "freeze" process since affinities are only tested once now. To make it competitive in terms of speed, we used quite a number of tricks that we will mention during the following explanation of the algorithm:

1. Lines up to 13 initialize the data structures. We can consider for now that the "simplified" argument is the empty set.



Function `Brute_Force_Improved`(G, $\mathcal{A}$, simplified, degree)

Data: Graph $G = (V, E)$, affinities $\mathcal{A}$ with weight function $w : \mathcal{A} \to \mathbb{N}$, subset of nodes already simplified, array containing the degree of each node, k number of colors.
Output: TRUE if G is greedy-k-colorable, FALSE otherwise.
Result: Graph G is conservatively coalesced.

```
1  stack ← simplified ;
2  move_related_worklist ← ∅ ;
3  simplify_worklist ← ∅ ;
4  hi_degree_worklist ← ∅ ;
5  Affs ← A ;
6  Function update_worklist(x) ;          /* Function to move x to the right worklist. */
7  begin
8      remove x from the worklist it belongs to;
9      if ∃⟨u, v⟩ ∈ Affs, u = x or v = x then move_related_worklist ←
       move_related_worklist ∪ {x} ;
10     else if degree[x] < k then simplify_worklist ← simplify_worklist ∪ {x} ;
11     else hi_degree_worklist ← hi_degree_worklist ∪ {x} ;
12 end
13 foreach x ∈ V \ simplified do update_worklist(x)
14 while TRUE do
15     if simplify_worklist ≠ ∅ /* Now, simplify the graph */ then
16         let x ∈ simplify_worklist;
17         simplify_worklist ← simplify_worklist \ {x} ;
18         foreach w neighbor of x do
19             degree[w] ← degree[w]-1 ;
20             if degree[w] = k-1 then update_worklist(w)
21         push x on stack ;
22     else if Affs ≠ ∅  /* All nodes are move-related or not simplifiable, try to coalesce */
       then
23         let ⟨x, y⟩ ∈ Affs ;
24         Affs ← Affs \ {⟨x, y⟩};
25         if x and y are neighbors /* cannot be coalesced */ then
26             update_worklist(x) ;
27             update_worklist(y) ;
28         else if Briggs_George_Coalescing (x,y) then
29             merge x and y into xy in G ;
30         else
31             merge x and y into xy in G ;
32             degree' ← copy of degree ;
33             if Brute_Force_Improved(G, ∅, stack, degree') = FALSE then
34                 un-merge x and y in G ;
35                 update_worklist(x) ;
36                 update_worklist(y) ;
37         if node xy exists /* i.e., x and y have been merged */ then
38             remove x and y from any worklist ;
39             update_worklist(xy) ;
40     else if hi_degree_worklist ≠ ∅ then return FALSE else return TRUE
```

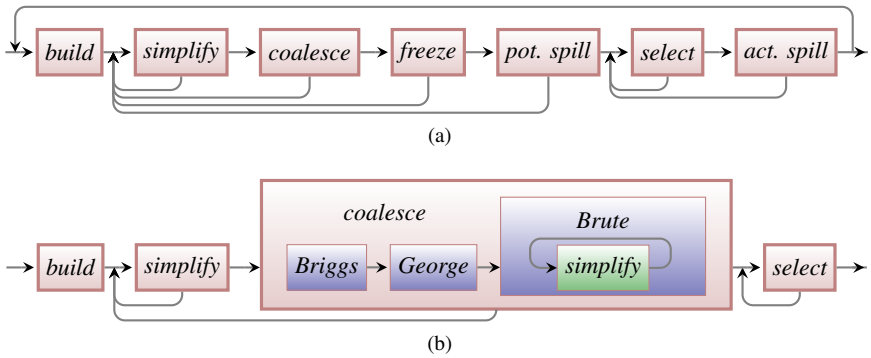


Figure 6.1: Comparison of: (a) the IRCscheme with (b) our "brute force improved" scheme.

2. The simplification of the graph is done from Line 15 to Line 21. This is the same as the "simplify" box of IRC.

3. Whenever no more node is simplifiable (usually because low-degree nodes are move-related), we enter the "coalesce" box, Line 22. We use local rules first: if Briggs's or George's rule applies at Line 28, there is no need to check an affinity with a brute-force test.

4. If Briggs's and George's rules fail, we could call the `Brute_Test` at Line 30, but this function is just a wrapper around Function `Is_kGreedy`, which in turn is just made of the "simplify" box of IRC. Since we already coded the simplification algorithm in `Brute_Force_Improved`, we preferred to do a recursive call, which appropriate parameters:

   - $\mathcal{A}$, the set of affinities passed, is empty. Hence, the recursively called function will never enter the "coalesce" box at Line 22: it is restricted to the "simplify" part and will then only check that the graph is greedy-k-colorable;
   - the stack of already simplified nodes is used as pre-simplified nodes to speed up the greedy test;
   - a copy of the array of degrees is passed to avoid recomputing them;
   - worklists of nodes are local to each function, hence they do not need to be updated after the recursive call returns (however, they need to be updated at the beginning of the recursive call, Line 13, because in that case no node is move-related anymore).

5. Finally, Line 40 tests whether the graph is completely simplified or not. If it is not, it means that either coalescing x and y leads to a non greedy-k-colorable graph (case of the recursive call), or that the initial graph G is not greedy-k-colorable (which should not happen).

Even if this algorithm seems a bit long and complicated, it is in fact simpler than IRC. We put the two algorithms one next to the other on Figure 6.1. Notice that in the IRC scheme, the "coalesce" box also contains Briggs's and George's rules. In our case,

we do not need to spill anymore, so the spill related boxes disappeared, and also do the last back edge, which is needed to rebuild the interference graph and start again if there was any spill. Also, the last coalescing test, the "brute" box, is only a duplication of the second leftmost box, i.e., the same simple simplify scheme.

We would like to stress out a few more points that are not directly apparent in the pseudo-code, but important in terms of speed of the resulting code, or easiness in the implementation.

- We always work with the original graph G, and not duplicates. It saves precious time at every recursive call. Function `Brute_Force_Improved` makes it possible by being not destructive: it is implemented in an IRC-like fashion using worklists, hence not requiring an actual removal of nodes in the graph.[2]
- When testing an affinity $\langle u, v \rangle$ using the recursive call, instead of actually merging u and v Line 31 before checking if the graph is still greedy-k-colorable, we use a trick on the degree of u, v and their neighbors to simulate the merge. Indeed, we found it was painful to perform an "un-merge" in the graph, and thought easier to decrease by one the degree of the common neighbors of u and v, and to increase the degree of u and v to be the degree that node uv would have.
- Finally, in the recursive call, it is possible to stop whenever uv (or, actually, u or v since they are not really merged) becomes a low-degree node. Indeed, if the graph was greedy-k-colorable, merging u and v only increases the degree of resulting node uv: the degree of all other nodes either decreases or stays the same. Hence, if uv becomes simplifiable, the whole graph is simplifiable.

These optimizations lead to a improved version of the `Brute_Test`, tuned for speed but with the same quality results. `Brute_Force_Improved` outperforms all previous conservative approaches with acceptable running times. It uses the same framework as the IRC, but with recursive calls, and is thus easy to plug in IRC, and not more complicated to implement standalone. It is approximately 2× slower than the basic IRC presented by Appel and George [2001] (without the speed improvements of Leung and George [1999b]) but reduces by a factor of 2 the cost of remaining affinities for the suite of graphs from the Coalescing Challenge [Appel and George, 2000]. Surprisingly, at the price of a 3× slow down (roughly), it is even a lot better by 15% than state-of-the-art optimistic coalescing, which contradicts the common belief. The details of these experiments will be explained is Section 6.5.

Brute-force coalescing is still an incremental strategy, meaning that the graph remains greedy-k-colorable after the coalescing of each particular affinity. We try to go even further in Section 6.2.2 with our "chordal-based" coalescing.

6.2.2 Chordal-based incremental coalescing

As previously mentioned, it is a limitation to require the graph to be greedy-k-colorable after each coalescing. Indeed, to get a greedy-k-colorable graph after coalescing an affinity, it may be needed to coalesce other affinities or even merge nodes not linked by any affinity. This fact was already known to George and Appel [1996] and used by Vegdahl [1999] in an actual framework. It is also used in the "optimistic+" algorithm (extended optimistic coalescing) of Park and Moon [2004]. However, these additional

[2]Note that, in the book by Appel [1998], the "move_related_worklist" is called "freeze_worklist." We find this appellation misleading and preferred using the first one.

merges are done blindly, without any guarantee that a coalescing is indeed enabled. Actually, for a given affinity, the real conservative coalescing problem is as formulated in Section 6.1: deciding if, after one particular coalescing, the graph is still k-colorable. This amounts to know if the resulting graph can become greedy-k-colorable thanks to some additional node merges.[3] Such a test can coalesce affinities beyond traditional conservative coalescing, and even nodes not linked by affinities. But this is best not to coalesce too many nodes so as to keep the maximum liberty for coalescing the rest of the affinities.

Such a conservative test can be checked in polynomial-time for a chordal graph, i.e., given a k-chordal graph and an affinity $\langle u, v\rangle$, we know in polynomial time if u and v can be merged while G stays k-colorable. If they can, we also know in polynomial time a k-coloring of G. We proved this fact in Chapter 5, but the proof was only conceptual. A limitation is that we do not know a good way to keep the graph chordal as we will explain later. In this section, we develop a linear-time algorithm based on the following ideas. In the proof of Theorem 5.8, the key is to consider the representation of a chordal graph as a family of subtrees of a tree. Then, checking if an affinity $\langle u, v\rangle$ can be coalesced works by searching for a "path" of subtrees between u and v on the tree. If it exists, such a path follow the smallest path *on the tree* linking u and v, denoted $P_{u,v}$. On $P_{u,v}$, the subtrees of the tree are intervals, hence we search for a "path" of *intervals* between u and v. There follows the three main ideas to find the set of intervals between u and v, and then search for a path in them:

First: it is easy to find the subtrees that are sub-intervals of $P_{u,v}$ by applying Chaitin et al.'s simplification scheme but forbidding u and v to be simplified. This has the effect of "pruning" the useless subtrees for finding a path between u and v.

Second: once only the "interesting" subtrees remain, a perfect elimination scheme gives an interval representation. It is then easy to search for a path between u and v in this interval graph.

Third: simplifying the nodes with minimum degree first, in this interval, defines a perfect elimination scheme.

Of course, the last point is not true in the general case, but in our particular case while working on an interval, it is true, as we will prove it in the next section.

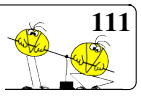
6.2.2.1 Two lemmas for chordal-based coalescing

Before explaining the complete algorithm based on the exposed ideas, we will first give two lemmas that will prove their correctness. They are based on a characterization of chordal graphs as graphs having a perfect elimination scheme (see Definition 2.19).

Lemma 6.1. *A chordal graph with only two simplicial nodes is an interval graph. Furthermore, any perfect elimination scheme gives an interval representation.*

Lemma 6.2. *Let G, with nodes $v_1, \ldots, v_n$, be a k-colorable chordal graph. If for all i, $1 < i < n$, the degree of v_i in G is at least k and is minimum in $G \setminus \{v_1, \ldots, v_{i-1}\}$, then $v_1, \ldots, v_n$ define a perfect elimination scheme for G.*

[3] If G is k-colorable, this is always possible since merging all nodes with the same color leads to a k-clique, which is greedy-k-colorable.



Lemma 6.1 will be used to validate the first two ideas, since only u and v can be simplicial vertices after the phase of prunning. Lemma 6.2 validates the third idea. We now prove these lemmas.

Proof of Lemma 6.1. Let $G = (V, E)$ be a chordal graph with only two simplicial nodes u and v. (We recall that a node is simplicial if its neighbors form a clique, see Definition 2.19.) A graph is chordal iff it has a perfect elimination scheme [Golumbic, 1980, Theorem 4.1], i.e., a particular order of nodes $v_1, \ldots, v_n$, such that the neighbors of v_i in $G \setminus \{v_1, \ldots, v_{i-1}\}$ form a clique. Furthermore, there is such a scheme with $v_1 = u$ and $v_n = v$. Indeed, a chordal graph has always at least two simplicial nodes and a subgraph of a chordal graph is chordal, therefore, we can first pick $v_1 = u$ and then always select a simplicial node $v_i \neq v$ to define a perfect elimination scheme starting with u and ending with v.

In this elimination scheme, for all $i > 1$, there exists $j < i$ such that $(v_i, v_j) \in E$. Indeed, if this is not the case, the neighbors of v_i in $G \setminus \{v_1, \ldots, v_{i-1}\}$ are the neighbors of v_i in G. Thus, v_i is simplicial in G, which is not possible unless $i = n$, as G has only two simplicial nodes, v_1 and v_n. For $i = n$, all neighbors v_j of v_n are of course such that $j < n$, unless $v(n)$ has no neighbor. But, in this case, $G \setminus \{v_n\}$, which is chordal, has two other simplicial nodes, thus G has at least 3 simplicial nodes.

Each node is thus neighbor in G of a node eliminated (simplified) before. We can prove more: if v_i and v_j, with $j < i$, are neighbors in G then v_i is also neighbor of v_k for all $j \leq k < i$. Indeed, suppose this is not the case. Let i be the smallest for which this property does not hold and let j be the largest such that $j + 1 < i$, with v_j neighbor of v_i but v_{j+1} is not. Also, v_{j+1} is a neighbor of v_j, otherwise i is not the smallest. But the neighbors of v_j in $G \setminus \{v_1, \ldots, v_{j-1}\}$, which include v_{j+1} and v_i, form a clique. Thus v_i is neighbor of v_{j+1}. Impossible.

With the last property, we can view the nodes v_i as points on a line, drawn from left to right by increasing i, and G can be interpreted as the interference graph of n intervals. Each v_i corresponds to an interval that ends at v_i and starts at v_j, for the smallest j such that v_j is neighbor of v_i. Indeed, for all $j \leq k < i$, the interval corresponding to v_k intersects the interval corresponding to v_i.

We can prove more formally that G is an interval graph. A chordal graph is an interval graph if its complement is a comparability graph [Golumbic, 1980], i.e., if there is an order $\prec$ on the nodes with the following property: for any three nodes x, y, z such that $x \prec y \prec z$, if $(x, y) \notin E$ and $(y, z) \notin E$, (i.e., they are both in the complement of G), then $(x, z) \notin E$ (it is also in the complement). This is true if $\prec$ is the perfect elimination scheme order, because if $x \prec y \prec z$ and $(x, z) \in E$, then $(y, z) \in E$, i.e., y is neighbor of z (as previously shown in the third paragraph of the proof). □

Proof of Lemma 6.2. As G is k-colorable, the size of any clique is at most k. Thus, only v_1 and v_n can be simplicial since all other nodes have degree $\geq k$, which would form a clique of size at least $k+1$. Furthermore, as G is chordal, Lemma 6.1 shows that it is actually an interval graph and any perfect elimination scheme that starts from v_1 and ends at v_n gives a representation of intervals. We now show that $v_1, \ldots, v_n$ define a perfect elimination scheme, i.e., selecting vertices with minimum degree provides such a scheme. If not, let i be the smallest such that v_i is not simplicial in $G \setminus \{v_1, \ldots, v_{i-1}\}$. Thus, for all $k < i$, v_k is simplicial in $G \setminus \{v_1, \ldots, v_{k-1}\}$. As $G \setminus \{v_1, \ldots, v_{i-1}\}$ is chordal, one can complete $v_1, \ldots, v_n$ into a perfect elimination scheme $w_1, \ldots, w_n$, such that $w_n = v_n$ and, for all $k < i$, $w_k = v_k$. Following Lemma 6.1, this elimination scheme

gives an interval representation, where the w_k are points on a line, drawn from left to right by increasing k, and where each w_k is the right end of an interval.

Consider the subgraph $H = G \setminus \{w_1, \ldots, w_{i-1}\} = G \setminus \{v_1, \ldots, v_{i-1}\}$. There are two cases, depending if v_i intersects w_i or not. If v_i is not a neighbor of w_i, it cannot be a neighbor of any node already simplified because $w_1, \ldots, w_n$ gives an interval representation. Therefore the degree of v_i in H is the degree of v_i in G, it is thus at least k, which is not minimum in H as the degree of w_i is at most $k-1$. Impossible. If v_i is a neighbor of w_i, any neighbor of w_i in H is a neighbor of v_i in H because w_i is simplicial in H hence its neighbors form a clique. Therefore, the degree of v_i in H is at least the degree of w_i in H and it cannot be minimum unless the degrees are equal. But, then, v_i has exactly the same neighbors as w_i in H hence is also simplicial. Impossible.

This proves that $v_1, \ldots, v_n$ is a perfect elimination scheme. □

6.2.2.2 Explaining the chordal-based algorithm

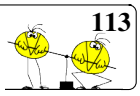
Using the two previous lemmas, we now explain the main idea on how our *chordal-based coalescing* works. Suppose one wants to coalesce u and v, two non-interfering nodes of a chordal graph G. First, G is simplified maximally (i.e., nodes with degree $< k$ are removed) without simplifying u and v. If no other node remains, u and v can be given the same color and coalescing them keeps the graph greedy-k-colorable (but not necessarily k-chordal). Otherwise, the two lemmas show that the remaining graph is an interval graph, for which one can easily compute a representation by removing nodes of smallest degree first. The rest of the algorithm looks for a "path" of non-interfering nodes between u and v in the interval graph, so that all these nodes can have the same color, as suggested in the proof of Theorem 5.8 in the previous chapter. The existence of such a path proves that u and v can be coalesced, along with all the nodes of the path, and G remains greedy-k-colorable. The non-existence of such a path proves that no matter how G is colored, u and v will always have a different color, hence the graph would not be k-colorable anymore if they are merged.

A pseudo-code for the "path searching" part of the chordal coalescing is given in Function `Chordal_Coalescing`, page 114. The whole algorithm is quite sophisticated, so we explain it in different steps. First we explain the general ideas that make it works. Then, we prove that `Chordal_Coalescing` returns TRUE iff u and v can be coalesced while G stays k-colorable. We then prove that in that case, the function finds a set of nodes to coalesce with u and v so that the interval graph on which it works stays a k-colorable interval graph. Finally, we will see how it can be integrated in Function `Brute_Force_Improved`.

General functioning of `Chordal_Coalescing`. Despite its name, this function expects in fact a more constrained graph than a chordal graph: an interval graph. We will use Lemma 6.1 later to prove that, from a chordal graph, one can get an interval representation by simplifying nodes of low degree first. Before explaining Function `Chordal_Coalescing` in details, we first give the general scheme of how it works when trying to coalesce two nodes x and y in an interval graph G:

1. Get an interval representation of G by simplifying nodes with smallest degree first.

2. Using this representation, propagate from x the information "this interval can/-cannot have the same color as x."



Function `Chordal_Coalescing`$(G, \langle x, y\rangle)$

Data: k-colorable *interval* graph $G = (V, E)$, affinity $\langle x, y\rangle$, all nodes other than x and y have degree $> k$.
Output: TRUE if coalescing $\langle x, y\rangle$ is conservative.
Result: If returning TRUE, x and y are merged, possibly with other nodes, so that G stays an interval graph.

```
1  if degree[x] > degree[y] then x, y ← y, x ; /* start with the smallest degree */
   /* Traverse the set of intervals from x to y. */
2  like_x[x] ← TRUE; alive ← {x} ; dummy_like_x ← FALSE;
3  just_removed ← ∅ ; like_x[just_removed] ← FALSE; ;   /* just for initialization */
4  nodes ← nodes \ {x} ;
5  repeat
6      let v ∈ nodes \ {y} with smallest degree ;
7      dummy_like_x ← dummy_like_x ∨ like_x[just_removed];
8      foreach w ∈ nodes neighbor of v do
9          degree[w] ← degree[w]-1 ;
10         if w ∉ alive then
11             alive ← alive ∪ {w} ;
12             like_x[w] ← dummy_like_x ;   /* w can have the same color as dummy */
13     if #alive = k then dummy_like_x ← FALSE ;   /* no dummy interval */
14     if #alive > k then return FALSE ;   /* cannot happen for interval graphs */
15     push v on stack ;
16     nodes ← nodes \ {v} ; alive ← alive \ {v} ; just_removed ← v ;
17 until nodes = {y} ;
18 if like_x[y] = FALSE then return FALSE;
   /* Else, construct the path linking x and y. */
19 path ← {y} ;
20 current ← y ;
21 while v ← pop stack, v ≠ x do
22     if v not a neighbor of current then
23         if like_x[v] then
24             path ← path ∪ {v} ;
25             current ← v ;
26 merge all v ∈ path into a single node in G ;   /* G stays an interval graph */
27 return TRUE
```

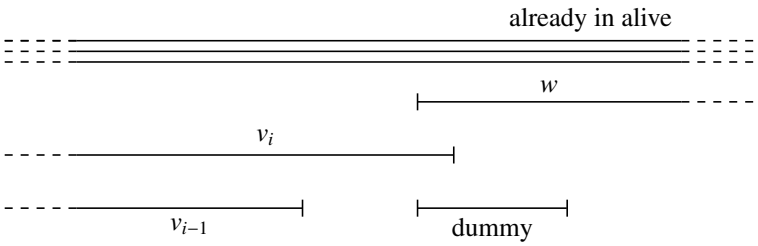


Figure 6.2: Position of relative intervals at iteration i.

3. Upon reaching y, there are two possibilities:
   - If "y cannot have the same color as x," it is impossible to coalesce x and y while staying k-colorable.
   - If "y can have the same color as x," then there is a path of intervals that can have the same color as x between x and y. One can be found by starting from y and choosing intervals in reverse order of simplification. Then, all nodes of this path can be merged with x and y and the final graph is greedy-k-colorable.

In `Chordal_Coalescing`, the first two steps are actually performed in only one pass, and "dummy" intervals are considered where there are less than k intervals at one point. The next two theorems are devoted to proving that the function works as expected by proving formally the steps described above.

Theorem 6.3. *Let G be a k-colorable graph and $\langle x, y\rangle$ an affinity. If G is an interval graph, then Function* `Chordal_Coalescing` *returns* TRUE *iff merging x and y leads to a k-colorable graph (not necessarily an interval graph).*

> Note: Line 14 is not needed for the correctness of the algorithm if G is a k-colorable interval graph. It is just needed for the extension to greedy-k-colorable graphs.

Proof. Let us concentrate on the decision part, from the first line up to Line 18.

Every node other than x and y has a degree $\geq k$. G is k-chordal hence x and y are simplicial vertices and their degree is at most $k-1$. Line 1 redefines x to be the one of smallest degree, hence, according to Lemma 6.2, an interval representation for G is obtained by simplifying x first, then each node $\neq y$ with minimum degree.

In the repeat loop, at the i-th iteration, v is the node v_i in Lemma 6.2, just_removed is v_{i-1} before being updated at Line 16 for the next iteration, and alive (updated at Line 11) contains all intervals alive at point v_i. Indeed, alive is completed by all neighbors w of v_i not already in alive. The corresponding intervals thus start just after v_{i-1} ends. Also, if at most $(k-1)$ intervals are alive at point v_i, one considers that a dummy interval is alive at this point. This situation is depicted in Figure 6.2, and in the algorithm, the presence of a dummy interval is emulated by using the variable dummy_like_x, which is set to TRUE or FALSE depending on whether the dummy interval could have the same color as x or not. If k intervals are alive at this point, there is no dummy interval and it is equivalent to say that the dummy cannot have the same color as x: dummy_like_x is set to FALSE (Line 13).

It is easy to see that G has a coloring such that x and y have the same color iff there is sequence of intervals, possibly including dummies, such that each interval

ends just before the next one starts, and the first interval is x, the last is y. Therefore, it is sufficient to propagate a flag, starting from x, from ends of intervals to starts of intervals and see if y can be reached. This is what the propagation of the variables like_x[w] and dummy_like_x does. If there is a coloring in which x and y have the same color, then the propagation from like_x[x] = TRUE reaches y. We now see the converse more formally.

Let us prove that, if like_x[w] is set to TRUE at iteration i (Line 12), the following holds: if there is a k-coloring of w and of all nodes simplified *later* (thus colored *before* in the select phase) and such that w and y have the same color, then one can build a coloring of G such that y and x have the same color. A similar property holds for the dummy interval at v_i as like_x[w] and dummy_like_x have the same value. Let us prove this property by induction on i, the implicit loop counter of the repeat loop. This is true for $x = v_1$ due to the initialization (before the repeat loop). Assume the property is true for all $j < i$. Let w be a node added in alive at iteration i and such that like_x[w] = TRUE (the same argument can be used for the dummy interval at v_i). Consider a coloring of all nodes simplified after w such that w and y have the same color. As G is greedy-k-colorable, this coloring can be extended, popping nodes from the stack, to all nodes from w down to v_i, without using the color of w because all these nodes interfere with w. If like_x[w] is TRUE because like_x[v_{i-1}] (resp. dummy_like_x) is TRUE, one can color v_{i-1} (resp. the dummy interval at v_{i-1}) with the color of w. Finally, by induction hypothesis at iteration $i-1$, we can extend the coloring to G such that x and y have the same color. □

We have proved that Function `Chordal_Coalescing`, up to Line 18, correctly answers whether G stays k-colorable if x and y are merged. The proof shows even more: it shows that we know, in the interval representation, all the nodes that can be colored with the same color as x and y.

It remains to prove that, with additional node merges, we can obtain a interval graph again. This is done by following a path from y to x when popping nodes from the stack in the while loop, Line 21, and we prove it now.

Theorem 6.4. *If x and y can be merged while G stays k-colorable, then Function* `Chordal_Coalescing` *correctly choses a set of nodes to merge with x and y so that the graph stays an interval graph.*

Proof. Since we already know which intervals can be colored with the same color as x and y. We need to prove that the while loop at Line 21 finds a path between x and y in these intervals. Indeed, at each step, the variable current defines the last interval of the path under construction, starting at y. Initially, it is true since it is set to y. When a node v is popped, if it is a neighbor of current, it cannot have the same color as y (current has the same color as y) hence cannot be part of the path. The first node not a neighbor of current, and that can have the same color as y is indeed set to the same color as y: it is added to the path. Finally, when x is popped, it cannot be a neighbor of current since the first part of the algorithm stated it can be colored with the same color as y, hence the same color as current.

But, where are the dummy intervals? In fact they are not needed in the path. If there is a gap between two consecutive nodes v and w in the path, there was at most $k-1$ intervals alive at each point of the gap, hence it was "filled" with dummies during the propagation phase. Since w can be colored like y, it means that all these dummies can also be colored like y, hence it is also true for v. So, dummy intervals are not required in the path between x and y.

Now, we say that merging all the nodes of the path amounts to form a complete interval from x to y, thus transforming the graph into another interval graph, with at most k live intervals, hence k-colorable. There is a subtlety since, to get an actual interval, one should have included dummies in the path and merged with them. But since the merge is immediate, this just means that the merged node would interfere also with all the neighbors of the dummy intervals, i.e., the other intervals alive at these points. But, these interferences already exist after the merging of the path without dummies, otherwise, one of these other intervals, say w, would be fully included in a portion covered by dummy intervals, which would all be colorable with the same color as x and y. Thus w would have been added to the path at Line 24. □

Up to now, we have proven that Function `Chordal_Coalescing`, when given as input an interval graph where only x and y can be simplified, can decide in linear time whether x and y can be coalesced while staying k-colorable, and, if they can, merges a set of nodes so that G stays an interval graph. There are two interesting questiors.

First, how do we apply the algorithm to k-chordal graphs?

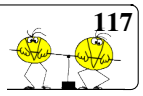
Second, does this algorithm also work with greedy-k-colorable graphs?

To answer the first question, it is tempting to use Lemma 6.1 to say that, after G as been maximally simplified into G' without simplifying x and y, G' can be fed to `Chordal_Coalescing`. Then, if x and y can be merged, the function merges nodes in G' along with x and y so that it stays an interval graph. Up to now, this reasoning is correct, but it would be false to say that adding back the simplified nodes to G' gives a chordal graph. Indeed, during the simplification, it can happen than nodes with a low degree "branch out" of G', i.e., are neighbors of nodes in G' but can nevertheless be simplified. If such a node has in G' only neighbors that are *not in the path* from x to y, it will not become a neighbor of the merged node xy, which can break the representation of G as subtrees of a tree, hence its chordality. We do not have a satisfying answer to the problem of staying in the k-chordal class with additional merges. So, the answer to the first question is:

> *By simplifying maximally a k-chordal graph, without simplifying x and y, and feeding the result to* `Chordal_Coalescing`*, we know whether x and y can be coalesced while G stays k-colorable but there is no guarantee that G stays k-chordal.*

Extension to greedy-k-colorable graphs. We now answer the second question, i.e., does `Chordal_Coalescing` work with greedy-k-colorable graphs? In fact, it does not provide an exact test as for chordal or interval graphs, but we can certify that, if the function does not return FALSE at Line 18, it is possible to color x and y with the same color. However, the converse is not true, i.e., even if there exists a coloring of G in which x and y have the same color, the function can return FALSE.

Proof. Suppose that G was first maximally simplified, before being given to Function `Chordal_Coalescing`. If G is only greedy-k-colorable, but not necessarily chordal, nodes are simplified in some order $v_1, \ldots, v_n$, and the repeat loop behaves as if G was an interval graph G' where each interval goes from v_j (supposing it is added in the set alive at iteration j) to v_i (supposing it is simplified at iteration i). This amounts to assume that all nodes in the set alive form a clique, even if some of them do not interfere. However,



there is no need to add these edges explicitly. The result of Chordal_Coalescing is then given with respect to this interval graph G', for which G is a subgraph. It is possible that G' is not k-colorable (Line 14) but if Chordal_Coalescing returns TRUE, it is safe to merge x and y. The additional merging of the nodes on the path then transforms G' into another interval graph and G into a subgraph of it, it is therefore greedy-k-colorable. Taking into account all nodes previously simplified before entering the function, the graph is still greedy-k-colorable. □

In this case, even if the test is less powerful than for k-chordal graphs, we at least have a guarantee that the final graph will stay greedy-k-colorable. This makes the chordal coalescing approach interesting in an incremental-like coalescing algorithm, where the graph needs to stay greedy-k-colorable at each step.

However, this is just a heuristic for greedy-k-colorable graphs that amounts to add interferences on the fly so that the graph "looks" chordal. Variants are certainly possible, with strategies to adding interferences, merge nodes, or maybe both, but we did not try to go further in this direction. However, what can easily be done is to select a path that, if possible, coalesces additional affinities and avoids merging nodes not related by affinities so as not to constrain too much the resulting graph. For the same reason, "dummy" intervals should be chosen when the path between x and y is built to avoid putting the constraints on actual intervals. It was not possible when working with interval graphs, as some interferences are required to stay in the interval graph class. However, greedy-k-colorable graphs stays greedy-k-colorable even if they have less interferences. In order to put the preference on dummy intervals, one should check, before adding v to the path at Line 24, if there was $< k$ variables in the alive set at this point (i.e., there was a dummy). This means the size of alive set at each step must be kept in memory, for instance in an array (of size at most n).

Using Chordal_Coalescing in Brute_Force_Improved. We have seen that the chordal-based coalescing can be used as a heuristic for greedy-k-colorable graphs. We now explain how to insert it in our existing "brute-force" scheme. Before giving the graph to Chordal_Coalescing, it is still possible to check if the coalescing can be done using Briggs's and George's rules, or the brute rule. Moreover, the graph must first be maximally simplified before the chordal coalescing can be applied. We will use the fact that, when the brute test returns FALSE and we would like to perform a "chordal test," the brute algorithm almost did a maximal simplification on the graph: the only difference is that x and y where already merged in the graph during the simplification.

In fact, this is not a problem. Maybe more nodes are simplified (since common neighbors of x and y have a smaller degree), but when the simplification is blocked in the brute test, all remaining nodes have degree $\geq k$. If we de-coalesce x and y, the degree of every node but x and y increases by one or stays the same; but the graph is then still subgraph G' of the original G. If G was chordal, G' also is, and it has at least two simplicial vertices. These can only by x and y, and their degree is then $< k$. Lemma 6.1 states this graph is an interval graph, and since every node but x and y have degree $\geq k$, the graph is maximally simplified which is what Function Chordal_Coalescing expects. If G was only greedy-k-colorable. It is of course still not a problem since the condition was that it must be maximally simplified. The same argument on the degree of nodes when de-coalescing x and y applies.

We just showed that the chordal-based coalescing can be easily integrated in our brute framework as follows: at Line 40 of Function Brute_Force_Improved instead

of returning FALSE, call instead Function Chordal_Coalescing.

6.2.2.3 Complexity and quality of chordal-based

In terms of complexity, chordal-based coalescing is similar to a complete simplification (with a return phase such as the classical "select" coloring phase). Thus, its complexity is similar, in order of magnitude, to brute-force coalescing. Also, used in complement to brute-force coalescing, its use does not increase the running time too much as most simplifications are already done. This algorithm can even be applied if the graph obtained after the first phase of simplification is not chordal, but only greedy-k-colorable.

In terms of quality of results, our experiments show that chordal-based coalescing does improve brute-force coalescing, but only slightly for the graphs of the Coalescing Challenge [Appel and George, 2000]. Section 6.5 discusses how far it is from optimality, thanks to an optimal ILP approach. In practice, because it is more complicated to implement, with only a marginal improvement, we believe it is maybe not worthwhile. However, this is the most advanced conservative algorithm proposed so far, and is also interesting from a graph theory point of view. There remain two questions. First, how to keep a chordal graph k-chordal? We explained previously that the problems comes from simplified nodes that "branch out" of the interval representation. Merging the path from x to y with these nodes would keep the representation as subtrees of a tree, but this seems difficult to modify the algorithm as it is now. Second, the remaining challenge would be to have an exact test on greedy-k-colorable graphs, as we left this problem open in the complexity study of Chapter 5.

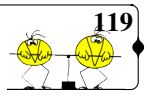
The next section aims at developing more advanced *optimistic* coalescing strategies.

6.3 De-coalescing after aggressive coalescing

Merging two nodes can transform a graph that is not greedy-k-colorable into a greedy-k-colorable graph, as was explained on Figure 5.6 in the previous chapter. This observation is the main motivation for aggressive coalescing: it may be more beneficial to first coalesce aggressively as many affinities as possible, then to try to undo, "de-coalesce" some coalescings if the graph is not greedy-k-colorable. But how to de-coalesce, i.e., *split* back affinities?

6.3.1 The existing strategy

We explained in the introduction of this chapter that, after an aggressive coalescing phase, Park and Moon [2004] proceed with the standard simplify and select phases. In the select phase, if the result of a merge cannot be colored—i.e., it was a "potential spill" and no color remains—it is split back into its original nodes. It is not safe to color all of them right away, even when it is possible, because this could prevent the coloring of nodes that are still on the stack, hence not colored yet. This is the case for common neighbors of the de-coalesced nodes, for which the degree is now bigger than when they were coalesced and simplified. However, if some de-coalesced nodes are colored with a unique color and the others discarded for now, the rest of the select phase can continue safely: hence Park and Moon choose heuristically: which nodes to color (by testing every possible configuration, assuming pessimistically that the other nodes will be spilled), which are spilled right away (those that, even completely de-coalesced, still have no color available), and the others, which are put at the bottom of the simplify stack, hence will be colored last during the rest of the select phase.



Appel and George [2001] pointed out that, if all nodes (except pre-colored nodes) have degree $< k$ initially, they can always be colored whatever the other colors. Based on this observation, they modified Park and Moon's de-coalescing algorithm,[4] but it is not clear from their explanations whether: (a) they some nodes with the same color, and the other nodes at the very end, as Park and Moon do, but with the guarantee that no spill occurs; or (b) they color all of them immediately, even if this can create subsequent de-coalescing, even of nodes that were not marked as potential spills. We tried to recreate their results, which were provided for a large collection of graphs in the Coalescing Challenge web page [Appel and George, 2000]; hence we developed several split strategies during the select phase. The common initial step is that when no color is found for a node, we split it back into its primitive nodes and compute some information as follows: let P be the set of such primitive nodes, we first compute the colors available for each of them and consider the set A_P of affinities $\langle x, y\rangle$ such that either both x and y belong to P, or one is in P and the other is already colored. Then we tried the three following strategies:[5]

1. *Appel-George type:* Consider all affinities $\langle x, y\rangle$ in A_P in decreasing order of weights and merge x and y if there exists a color suitable for both x and y.

2. *Park-Moon type:* Select a color c that maximizes the sum of the weights of the affinities $\langle x, y\rangle \in A_P$ such that x and y can be colored with c. Merge the corresponding nodes, color the resulting node with c, and put all other nodes aside and color them at the end of the select phase, using a biased[6] coloring.

3. *Iterated Park-Moon:* Select a color c and merge nodes as in the second strategy. Repeat the color selection for the remaining affinities in A_P until $A_P = \emptyset$.

Surprisingly, none of these approaches give results close to the optimistic version of Appel and George. Also, even though it seemed to us that Strategy 1 approaches the most their strategy, it is the worst of all three versions, certainly because it does not have a global view on the affinities within the split node. Nevertheless, we point out that the three of them are guaranteed to be spill-free only if all initial nodes have degree $< k$, and it is not clear how to adapt these heuristics to general greedy-k-colorable graphs. Furthermore, even if the initial degrees are $< k$, except pre-colored nodes, a problem may occur with Strategies 1 and 3 if pre-colored nodes can be simplified as normal nodes can (as we do). As several colors are given right away to the different primitive nodes, a simplified node might become not colorable. The node can then be split into its primitive nodes, all of degree $< k$, *unless* one of them is pre-colored in which case the algorithm would fail! Luckily, this potential problem never occurred in any of the 474 graphs of the Coalescing Challenge. In addition to these applicability limitations, a weakness of these three approaches in terms of quality of results is that, instead of de-coalescing affinities, possibly one by one, many affinities are de-coalesced, even if not needed, when one node is split back. As this process is done in the coloring phase, only a form of biased coloring can help re-coalescing these useless de-coalesced affinities and it is not clear how to use classical conservative coalescing techniques.

For these reasons, we prefer to de-coalesce based on the graph structure itself, not on the particular order of nodes in the stack. An interesting side-effect of this approach

[4] The first non-journal version of Park and Moon [1998] was published six years before the journal one.

[5] Note that both Strategy 1 and 3 can lead to de-coalescing nodes which were not marked as "potential spill," possibly provoking a cascade effect.

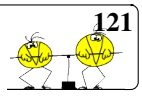
[6] When coloring a node, choose the color that maximizes the coalescing with its neighbors.

is that the final graph is then greedy-k-colorable and that we can use conservative techniques afterwards to improve our de-coalescing phase.

6.3.2 Our approach

As in Park and Moon optimistic coalescing, we start with a naïve aggressive phase. This phase consists only in considering affinities one by one and coalescing them if the two extremities do not interfere. After this phase, we use the de-coalescing phase described in Function `De-coalescing`, page 123. It applies to any greedy-k-colorable graph and produces a greedy-k-colorable graph. After aggressive coalescing, some affinities are de-coalesced as long as the graph is not greedy-k-colorable. These affinities are selected as follows. During the check for greedy-k-colorability (using Function `Is_kGreedy`, see page 21), if all nodes of the current subgraph have degree $\geq k$, the cheapest node to de-coalesce is chosen and split into two nodes. It may be necessary to de-coalesce several affinities to get two separated nodes. And, if considering a coalesced node and the graph of the affinities coalesced inside this node, there is *a priori* no restriction on the shape of the graph. Such a graph is presented on Figure 6.3. To find the cheapest set of affinities to de-coalesce, we need to find the cheapest set of affinities that disconnect the coalesced node. We use a min-cut algorithm on the affinities coalesced in the node, but simpler approaches are possible. To choose quickly which node to de-coalesce, each node is given a lower bound of the cost of its de-coalescing, i.e., the min-cut cost. Currently, this bound is set to the smallest affinity weight coalesced in the node. After de-coalescing, the simplify phase continues until the graph becomes empty or another de-coalescing is necessary. This way, first we avoid de-coalescing an affinity in the area where it does not help, i.e., among the nodes already simplified; second, we give up coalescing the cheapest affinities first. However, as in the Park and Moon approach, de-coalescing an affinity can increase the degree of nodes already simplified, thus, in general, we need to perform several passes—at most 3 in practice for Appel and George's graphs—until no de-coalescing is done, which ensures the graph is greedy-k-colorable.

As we will show in Section 6.5, our de-coalescing scheme alone is as good as the state-of-the-art optimistic coalescing algorithm. This is of course only true in terms of coalesced affinities as we compare algorithms that do not spill at all, while the original Park and Moon's algorithm also include a spilling strategy. However, it has two strong advantages. First, it can be applied to any greedy-k-colorable graph, without requiring any spill. Second, as it is not intermixed with coloring (the select phase), it can be followed by some conservative coalescing to clean up possible useless de-coalescings: then, even a simple conservative coalescing such as Briggs's and George's rules improves the results by 8%. To our surprise, however, it does not equal the quality of our "chordal-based" coalescing, not even the quality of the "brute-force" coalescing. The problem may come from our way to choose nodes to de-coalesce. But it is hard to define a good indicator of the benefit and cost of a de-coalescing, so as to guide the node selection. Also, it is very likely that some bad decisions are made even earlier, i.e., by the aggressive phase—for example, coalescing an expensive affinity that prevents the coalescing of many cheap ones—which are then difficult to repair by a greedy de-coalescing. Such considerations will be discussed heavily in the experimental section.



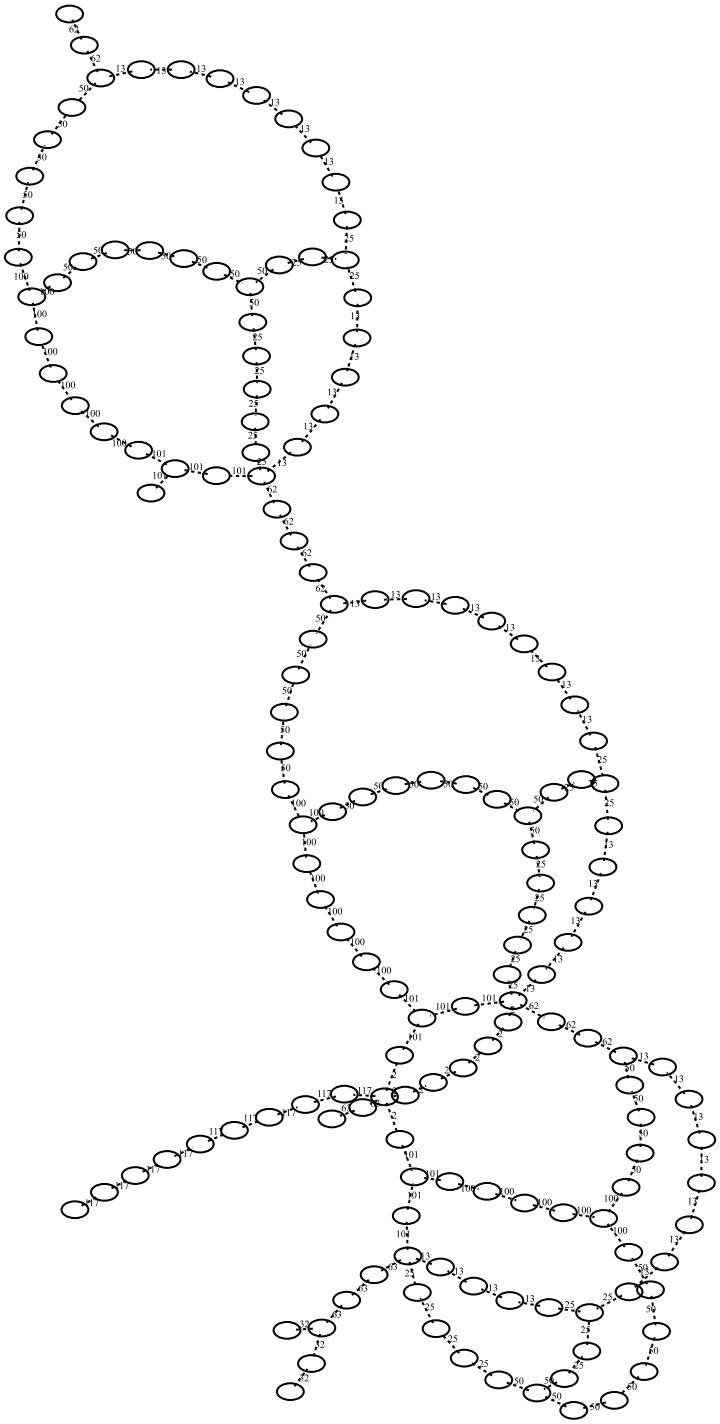


Figure 6.3: A graph of affinities coalesced in a coalesced node.

Function `De-coalescing`$(G, \mathcal{A})$

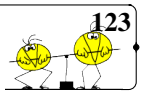
```
   Data: Graph G = (V, E), set of affinities A with weight function w : A → ℕ
 1 repeat
 2     nodes ← V ; stack = ∅ ;
 3     some_de-coalescing ← FALSE;
 4     while nodes ≠ ∅ do
 5         if ∃v ∈ nodes | degree[v] < k then
 6             nodes ← nodes \ {v} ; push v on stack ;                    /* Simplify v */
 7             foreach w neighbor of v do degree[w] ← degree[w]-1 ;
 8         else
 9             v ← get_min_cost(nodes) ;  /* coalesced non-simplified node, with smallest
               cost to de-coalesce */
10             ⟨x, y⟩ ← affinity in v with smallest weight ;
11             G ← de-coalesce_min_cut(G, ⟨x, y⟩) ;          /* de-coalesce x and y using a
               min_cut on the affinities of v */
12             nodes ← (nodes \ {v}) ∪ {x, y} ;
13             foreach w ∈ {x, y, neighbors of x and y} do update degree[w];
14             some_de-coalescing ← TRUE;
15 until some_de-coalescing = FALSE ;
```

6.4 Optimal rules for aggressive and conservative coalescing

The graphs from the Coalescing Challenge of Appel and George [2000] are very particular. In order to perform their optimal spill, the authors needed to insert a lot of splitting points, i.e., everywhere they could: between every two instructions. An example of such a graph is given on Figure 6.4: because of the splitting, it is composed of only cliques of size six or less, but for the pre-colored nodes (machine registers), which have more neighbors. These cliques correspond to instructions of the program.[7] Affinities link the nodes that correspond to the same variable split into hundreds of tiny live-ranges. Hence there are a lot of nodes with exactly two affinities: one towards the "next" clique, and one towards the "previous" one. The structure of the initial program can be visually guessed from the shape of the graph, with "chains" of cliques corresponding to instructions in a basic block, or branches corresponding to conditional statements, as in the zoomed part of the figure.

A set of affinities between two cliques corresponds to a program point, i.e., a place where shuffle code can be inserted if the colors do not match. Among these points, some are obviously not useful in the context of coalescing. A point is said to be "unnecessary" if there exists an optimal coalescing solution in which all the affinities of this point are coalesced. We wanted a way to find, in Appel and George's graphs, the maximum number of unnecessary split points so that we could coalesce the corresponding affinities right away. This would decrease the size of the graph and guide the

[7] We are still a bit puzzled by this fact. We were expecting the instructions to be represented by two intersecting cliques: the variables *live_in* of the instruction and the ones *live_out*. In their article, Appel and George [2001] do sometimes have structures like these, because of register constraints imposing an argument and result to reside in the same register. However, they still claim that, in the general case, there are only cliques, maybe because they consider it is always possible to repair the coloring around instructions without register constraints.



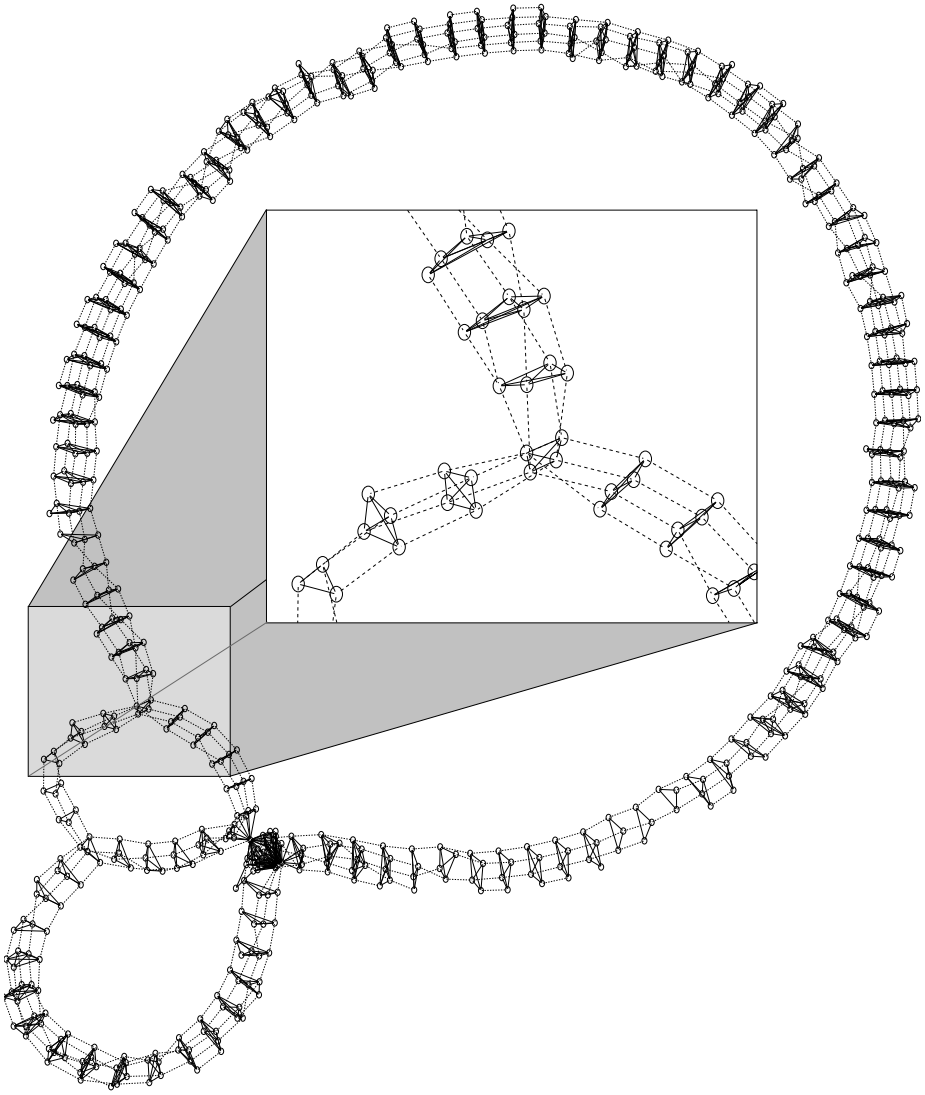


Figure 6.4: Graph #001 of the Coalescing Challenge.

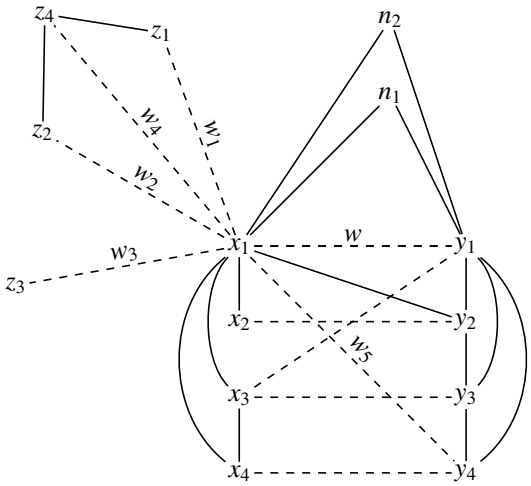


Figure 6.5: Optimal "clique" rule.

coalescing heuristics.

There is an easy condition under which it is safe—from an optimal point of view—to coalesce an affinity: the idea is that, given an affinity $\langle u, v\rangle$, if its cost is greater than the sum of all other affinities involving u, and if every neighbor of u is also neighbor of v, it is always safe to choose to coalesce $\langle u, v\rangle$ in terms of cost of the coalescing. This idea is close to George's rule, but the condition on the weights guarantees optimality. We will explain more formally this idea and extend it so that, in particular, it can cope with affinities of unnecessary split points in the graphs from the Coalescing Challenge.

6.4.1 The optimal "clique" rule

George's rule says that if all neighbors of x of significant degree—i.e., $\geq k$—are also neighbors of y, one can coalesce safely x and y—i.e., the graph will remain greedy-k-colorable. It is easy to modify the rule as follows: if all neighbors of x are also neighbors of y (not just the high-degree ones), and the weight of $\langle x, y\rangle$ is greater that the sum of the weights of all the other affinities involving x, it is safe to coalesce $\langle x, y\rangle$, i.e., there exists an optimal solution in which x and y are coalesced. Indeed, suppose that x and y have a different color in col, an optimal solution: $col(x) \neq col(y)$. It is possible to change $col(x)$ so that it is the same as y, and the overall cost of affinities not coalesced decreases. We will not prove this more formally here as we will provide a similar demonstration for the more general "clique" rule.

This "optimal" version of George's rule does not allow the coalescing of unnecessary split points. Indeed, for such points, there are two cliques corresponding to the instructions before and after the point: X containing variables $x_1, \ldots, x_n$ and Y containing $y_1, \ldots, y_n$, and an affinity $\langle x_i, y_i\rangle$ for $1 \leq i \leq n$. The rule cannot coalesce any of these affinities because every x_i has neighbors that are not neighbors of y_i: all the x_j with $j \neq i$! So, unless $n = 1$, this optimal rule is useless.

However, we point out that coalescing all $\langle x_i, y_i\rangle$ *at the same time* is possible. The conditions would then be the following. Consider two cliques X and Y such that all neighbors of X are also neighbors of Y. If, for each affinity $\langle x_i, y_i\rangle$, the weight of this affinity is greater that the sum of the weights of all the other affinities involving x_i, it

is safe to coalesce at the same time all the affinities $\langle x_i, y_i \rangle$, i.e., there is an optimal solution in which all $\langle x_i, y_i \rangle$ are coalesced.

In fact, the conditions can be relaxed, so that X and Y do not need to be cliques: X being structurally a subgraph of Y is sufficient (see the second point in the definition).

Definition: Let us define two sets of nodes X and Y as follows. Let $X = \{x_1, x_2 \dots x_n\}$ and $Y = \{y_1, y_2, \dots y_n\}$ such that:

- There is an affinity between each x_i and y_i: $\forall i, \exists \langle x_i, y_i \rangle \in \mathcal{A}$ (and $(x_i, y_i) \notin E$)
- X is structurally a sub-graph of Y: $\forall i, j, (x_i, x_j) \in E \implies (y_i, y_j) \in E$
- Every neighbor of X is neighbor of Y: $\forall i, (x_i, z) \in E$ with $z \notin X \implies (y_i, z) \in E$

Theorem 6.5 ("Clique" rule)**.** *Given two sets X and Y as defined above, there exists a conservative coalescing in which every affinity of type $\langle x_i, y_i \rangle$ is coalesced, i.e., there exists a coloring col in which $\forall 1 \leq i \leq n$, $col(x_i) = col(y_i)$.*

Moreover, there exists one such coalescing that is optimal if:

$$\forall x_i \in X,\ w\langle x_i, y_i \rangle \geq \sum_{\langle x_i, z \rangle,\ z \neq y_i} w\langle x_i, z \rangle$$

We will first prove that it is feasible to coalesce all $\langle x_i, y_i \rangle$, then we will prove that the constraint on the weights of the affinities is sufficient to make it safe to coalesce them.

Proof of feasibility. Let us define G_{coal}, the graph obtained from G by merging every node x_i with the node y_i. Since the greedy-k-colorable property depends only on the interference structure of the graph, let us prove that G_{coal} is a subgraph of G.[8] It is sufficient to prove that any merge does not create any new interference between nodes of $G \setminus X$. Any such new interference would be created by merging an affinity $\langle x_i, y_i \rangle$ with $(x_i, z) \in E$. Let us examine the different cases:

- if $z \in X$, then $z = x_j$ and by definition (y_i, y_j) already exists;
- if $z \in Y$, $z = y_j$, and $j \neq i$, hence (y_i, y_j) already exists;
- else, (y_i, z) already exists by definition of X and Y.

Hence the coalesced graph is a subgraph of the initial graph, which proves it is greedy-k-colorable if G was greedy-k-colorable □

Now we will prove that the constraint given on the weights of the $\langle x_i, y_i \rangle$ guarantee that there exists an optimal solution in which these affinities are coalesced.

Proof of optimality. The cost function of a coloring *col* is defined as :

$$\hat{c}(col) = \sum_{\langle x,y \rangle \in \mathcal{A}} \begin{cases} 0 & \text{if } col(x) = col(y) \\ w\langle x, y \rangle & \text{otherwise} \end{cases}$$

Hence, we will be looking for a coalescing of minimum cost. Let *col* be an optimal coloring—a coalescing of smallest cost—of G and $\overline{X} \subseteq X$ such that: $\forall x_i \in \overline{X},\ col(x_i) \neq$

[8] In fact, $G_{coal} = G \setminus X$

$col(y_i)$ and $\forall x_i \in X \setminus \overline{X},\ col(x_i) = col(y_i)$. Let col' be the coloring such that $\forall x_i \in \overline{X},\ col'(x_i) = col(y_i)$ and $\forall z \notin \overline{X},\ col'(z) = col(z)$.[9] col' is a valid coloring: if $x_i \in \overline{X}$ and z is a neighbor of x_i, either $z \notin X$ hence z is neighbor of y_i, or $z \in X$ hence $z = x_j$ and $col'(x_j) = col(y_j) \neq col(y_i) = col'(x_i)$. The cost of col' is the following:

$$\begin{aligned} \hat{c}(col') &= \hat{c}(col) + \text{affinities de-coalesced} - \text{affinities coalesced} \\ &= \hat{c}(col) + D - C \end{aligned}$$

We want a condition that ensures $\hat{c}(col') \leq \hat{c}(col)$, hence a condition under which $D \leq C$. Since $\{\langle x_i, y_i\rangle \mid x_i \in \overline{X}$ is a subset of the newly coalesced affinities,

$$\sum_{x_i \in \overline{X}} w\langle x_i, y_i\rangle \leq \text{weight of newly coalesced affinities}$$

which gives us a lower bound: $lb(C) \leq C$.

As for D, we will consider the worst case. Let $friends(z)$ be the set of affinity-neighbors of z, i.e., $\forall z' \in friends(z)$, $\langle z, z'\rangle \in \mathcal{A}$. For a given x_i, the worst case happens when x_i was optimally coalesced with the biggest subset (in terms of weight) of $friends(x_i) \setminus \{y_i\}$, and is now de-coalesced from all of them. The best coalescing of x_i is the biggest weighted interference-independent set of $friends(x_i) \setminus \{y_i\}$, denoted $IS(x_i)$, where the weight of z in this set is $w\langle x_i, z\rangle$.

$$D \quad \leq \quad \text{worst-case de-coalescing of all } x_i \in \overline{X} \quad \leq \quad \sum_{x_i \in \overline{X}} IS(x_i)$$

which gives us $ub(D)$, an upper bound on D. Hence $lb(C) \geq ub(D)$ is a sufficient condition since:

$$C \geq lb(C) \geq ub(D) \geq D$$

So the condition writes:

$$\sum_{x_i \in \overline{X}} w\langle x_i, y_i\rangle \quad \geq \quad \sum_{x_i \in \overline{X}} IS(x_i)$$

This is ensured by the following condition:

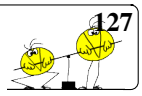
$$\forall x_i \in \overline{X},\ w\langle x_i, y_i\rangle \quad \geq \quad IS(x_i)$$

Since $\overline{X}$ is defined only for a given optimal coloring solution, and can be any subset of X, this condition must be ensured for all $x_i \in X$. An easier condition is found by using an upper bound of $IS(x_i)$ such as:

$$\forall x_i \in X,\ w\langle x_i, y_i\rangle \quad \geq \quad \sum_{z \in friends(x_i) \setminus \{y_i\}} w(x_i, z)$$

Since an independent set contains at most all the elements of the initial set, and X contains all elements of $\overline{X}$, this last condition clearly implies the previous one. □

[9] The colors of the x_i that do not match those of the y_i are changed so that they do.



It is easy to improve this naïve condition with a simple condition such as, if $z, z' \in$ *friends*(x_i) and $(z, z') \in E$, then add only $\max(w\langle x_i, z\rangle, w\langle x_i, z'\rangle)$ to the sum instead of the two weights.

We now give a pseudo-code for the clique rule, Function `Clique_rule`, page 129. The idea is to build incrementally the X and Y sets, while checking if affinities fulfill the conditions. We start from an affinity $\langle x, y\rangle$, while we still do not know the sets X and Y. We know from the clique rule that X must contain every neighbor of x not neighbor of y. These must have an affinity with neighbors of y (that are not already "chosen" by another node of X), else the clique rule cannot apply (Line 15). These neighbors must also fulfill the clique rule, so they are added to the "to_test" set. Finally, the check on the weight of $\langle x, y\rangle$ is performed and, if satisfied, the x node is declared "OK" and the process can be iterated with the remaining affinities of the test list. If every affinity fulfill the weight condition, the "X_OK" list after the loop contains exactly the X set of the clique rule (and the "Y_set" list contains Y), and affinities of the "to_coalesce" list can be safely coalesced.

6.4.2 The "terminal" rules

We will now present another case of optimality. The idea is that whenever there are two affinities $\langle x, y\rangle$ and $\langle x, z\rangle$ and an interference $(y, z) \in E$, it will never be possible to coalesce both affinities. Then, if either y or z is what we call a "terminal" node, i.e., it has only one interference and only one affinity, then it will always be possible to coalesce one of the two affinities, hence it is possible to "simplify" the affinity. Indeed, the idea is that, for instance if z is terminal, as in Figure 6.6, the choice whether $\langle x, z\rangle$ will be coalesced or not can be *postponed* until we know if $\langle x, y\rangle$ is. To do that, the weight of $\langle x, y\rangle$ can be changed to reflect the fact that coalescing it will prevent the coalescing of $\langle x, z\rangle$. As such, affinities can become of *negative weight*, meaning it is better *not to coalesce* these affinities.

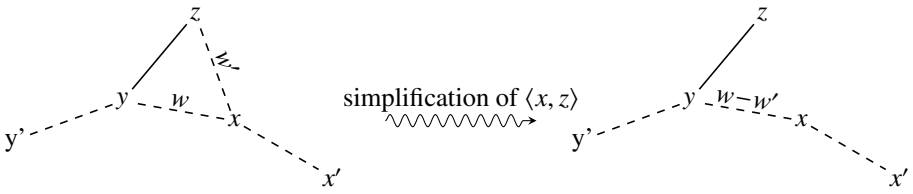


Figure 6.6: Simplify affinity $\langle x, z\rangle$ if z is terminal.

Definition 6.6. We call a node z *terminal* iff it has exactly one affinity $\langle x, z\rangle$ and one interference $(y, z) \in E$, with $x \neq y$.

It should be noted that nodes with only one affinity (and no interference) can be coalesced right away, and nodes with only one interference (and no affinity) are not of much interest in terms of coalescing.

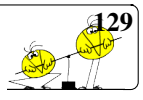
Theorem 6.7 ("Terminal" rule)**.** *Let x, y and z be such that y and z interfere, x has two affinities with y and z:* $\langle x, y\rangle$ *and* $\langle x, z\rangle$ *of weights w and w′ in G. If z is terminal, let G′ be the graph obtained by removing* $\langle x, z\rangle$ *in G and changing the weight of* $\langle x, y\rangle$ *to* $w - w'$*. If* $w' > 0$ *and col′ is an optimal conservative coalescing solution for G′ then:*

- *if* $col'(x) = col'(y)$*, then the coloring* $col = col'$ *is an optimal coalescing solution for G.*

Function `Clique_rule`$(G, \langle u, v\rangle)$

Data: Interference graph $G = (V, E)$, affinity $\langle u, v\rangle$.
Output: TRUE if coalescing $\langle u, v\rangle$ can be done by the clique rule.
Result: If returning TRUE, all affinities of the clique rule are merged in G.

```
   /* Initialize data.                                                          */
 1 X_OK ← ∅ ;
 2 Y_set ← ∅ ;
 3 to_coalesce ← ∅;
 4 to_test ← {⟨u, v⟩} ;
 5 while to_test ≠ ∅ do
 6     let ⟨x, y⟩ ← pop to_test;
 7     check ← neighbors of x \ neighbors of y;
 8     check ← check \ X_OK;
 9     while check ≠ ∅ do
10         let v ← pop check;
11         if ∃w ∈ {neighbors of y} \ Y_set such that ⟨v, w⟩ exists then
               /* Affinities between neighbors of x and neighbors of y must be tested. */
12             to_test ← to_test ∪ {⟨v, w⟩};
13             Y_set ← Y_set ∪ {w} ;
14         else
               /* One neighbor of x has no affinity with a neighbor of y.    */
15             return FALSE;
       /* Check the affinity has a larger cost than the sum of all other affinities of x. */
16     sum ← 0;
17     foreach ⟨x, y′⟩ | y′ ≠ y do
18         sum ← sum + abs(weight⟨x, y′⟩);
19     if sum > weight⟨x, y⟩ then
20         return FALSE
21     else
22         X_OK ← X_OK ∪ {x};
23         to_coalesce ← to_coalesce ∪ ⟨x, y⟩;
24 foreach ⟨x, y⟩ ∈ to_coalesce do
25     merge x and y in G ;
26 return TRUE
```

- *if* $col'(x) \neq col'(y)$*, then the coloring* col *such that* $col(z) = col'(x)$ *and* $col(u) = col'(u)$ *for* $u \neq z$ *is an optimal coalescing solution for* G*.*

Note that we impose the constraint that w' must be strictly greater than zero. Indeed, if $w' \leq 0$, it is best not to coalesce x and z. Since z has only one coloring constraint, it is always possible not to coalesce $\langle x, z\rangle$ if there is at least three colors. In this case, $\langle x, z\rangle$ can be replaced by an interference in the graph. This protects us from a perverse effect where both w and w' would be negative and $w - w'$ becomes positive: this would trick the coalescing algorithm into thinking it is good to coalesce $\langle x, y\rangle$ while it is of course better not to do it (if possible).

Proof. Since z is terminal, the only coloring constraint of z is having a color different that the one of y. In the first case $col'(z) \neq col'(y)$ since col' is a valid coloring of G', hence it is also a valid coloring of G. In the second case, $col(z) = col'(x) \neq col'(y)$ hence col is a valid coloring of G.

Now, for the optimality. Let us compute the cost of the solution col on G, $\hat{c}(col)(G)$, depending on $\hat{c}(col')(G')$. In the first case, where $col'(x) = col'(y)$, $\hat{c}(col)(G) = \hat{c}(col')(G) = \hat{c}(col')(G') + w'$ since every affinity in G is in the same state as in G' except $\langle x, z\rangle$, which is not coalesced. In the second case, $\hat{c}(col)(G) = \hat{c}(col')(G') - (w - w') + w = \hat{c}(col')(G') + w'$ since the affinity $\langle x, z\rangle$ is coalesced hence does not modify the cost, but the affinity $\langle x, y\rangle$ which is not coalesced has a different cost in G' than in G. Hence, in both cases,

$$\hat{c}(col)(G) = \hat{c}(col')(G') + w'$$

Consider col_{opt} an optimal solution. There are two cases depending on whether col_{opt} chooses to coalesce $\langle x, y\rangle$ or $\langle x, z\rangle$. In both cases, col_{opt} is obviously valid for G'. Let us compute its cost for G' depending on its cost on G:

- if $col_{opt}(x) = col_{opt}(y)$, then $\langle x, z\rangle$ is not coalesced, but does not exist in G':

$$\hat{c}(col_{opt})(G') = \hat{c}(col_{opt})(G) - w'$$

- if $col_{opt}(x) \neq col_{opt}(y)$, then $\langle x, y\rangle$ is not coalesced:

$$\begin{aligned} \hat{c}(col_{opt})(G') &= \hat{c}(col_{opt})(G) - w + (w - w') \\ &= \hat{c}(col_{opt})(G) - w' \end{aligned}$$

Suppose now that col is not optimal for G, then

$$\begin{aligned} \hat{c}(col)(G) &> \hat{c}(col_{opt})(G) \\ \hat{c}(col')(G') + w' &> \hat{c}(col_{opt})(G') + w' \\ \hat{c}(col')(G') &> \hat{c}(col_{opt})(G') \end{aligned}$$

This contradicts the optimality of col' in G'. Hence col is an optimal coalescing for G. □

When looking as particular graphs, we observed another case involving terminal nodes. This is the case where both y and z are terminal nodes:

Corollary 6.8 ("Double terminal" rule)**.** *If y and z are adjacent and both terminal edges, with affinities $\langle y, x\rangle$ and $\langle z, x'\rangle$, it is optimal to coalesce the affinity of greatest cost if it is greater than zero.*

> Note: of course, if the two affinities have negative weights, none of them will be coalesced in an optimal solution (if there is at least three colors).

Proof. Since y and z are terminal, they are disconnected from the rest of the graph in terms of interferences. Hence, whatever the coloring of the rest of the graph, the only restriction on these nodes is that they must have a different color, hence at least one of the two nodes can have the same color as its affinity-neighbor (in fact, both of them can unless $col(x) = col(x')$). Hence, in any coalescing solution, at least one of the two affinities can be coalesced. Moreover, if only one can be in the optimal solution, it is best to choose the biggest one. □

In a sense, the "terminal" rule is not actually an optimal rule since it does not optimally coalesce an affinity, but it still provides a way to simplify the problem without giving up the optimality of the final solution. Using Theorem 6.7, one can devise a strategy where affinities involved with a terminal node are simplified from the graph, i.e., removed but placed on a stack. If the remaining graph can be optimally coalesced, possibly using the "clique" or the "terminal" and "double terminal" rules again, the simplified affinities are then popped from the stack (i.e., in the reverse order of their simplification) and coalesced if possible. Then the final solution is also optimal. It that case one should take care that the terminal rule can create affinities with negative weights. Hence the condition on the weights for the clique rule should sum the *absolute weights* of the other affinities.[10] If, during this strategy, the remaining graph cannot be optimally coalesced, it can still be interesting to use then a heuristic, and pop the affinities simplified by the terminal rule at the end.

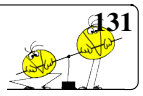
However, as shown by the experiments in Section 6.5, while the "clique" rule is really helpful, the conditions for the the "terminal" and "double terminal" rules to apply were never met in any of the 474 graphs of the Coalescing Challenge during our conservative coalescing tests. Indeed, the condition that a node should have only one interfering neighbor is very strong, and maybe the terminal rules should be tested after some simplifications of nodes. Still, they are of some use in a purely aggressive strategy, as explained in the next section and shown by our experiments, Section 6.5.5.2.

6.4.3 Using the optimal rules for aggressive coalescing

The optimal "clique" and "terminal" rules explained in the previous sections can be applied for conservative coalescing as well as for aggressive coalescing. But for aggressive coalescing, they can be relaxed as in this case there is no constraint on the number of colors to be used. This means that the graph can be partitioned in different "webs," i.e., components connected by affinities. We use the term "web" as, in SSA terminology, a *web* is a set of variables linked by ϕ-functions.[11] Figure 6.7 shows a different way of representing Appel and George's graphs which emphasizes affinity webs instead of cliques.

[10] The best possible coalescing for a node is then to be coalesced with all its affinity-neighbors of positive weight, and with none of its affinity-neighbors of negative weight. The worst case is the contrary.

[11] For instance, under conventional SSA (*c*SSA), all variables of a web can be renamed with the same name, i.e, coalesced. But we are not under cSSA in our case.



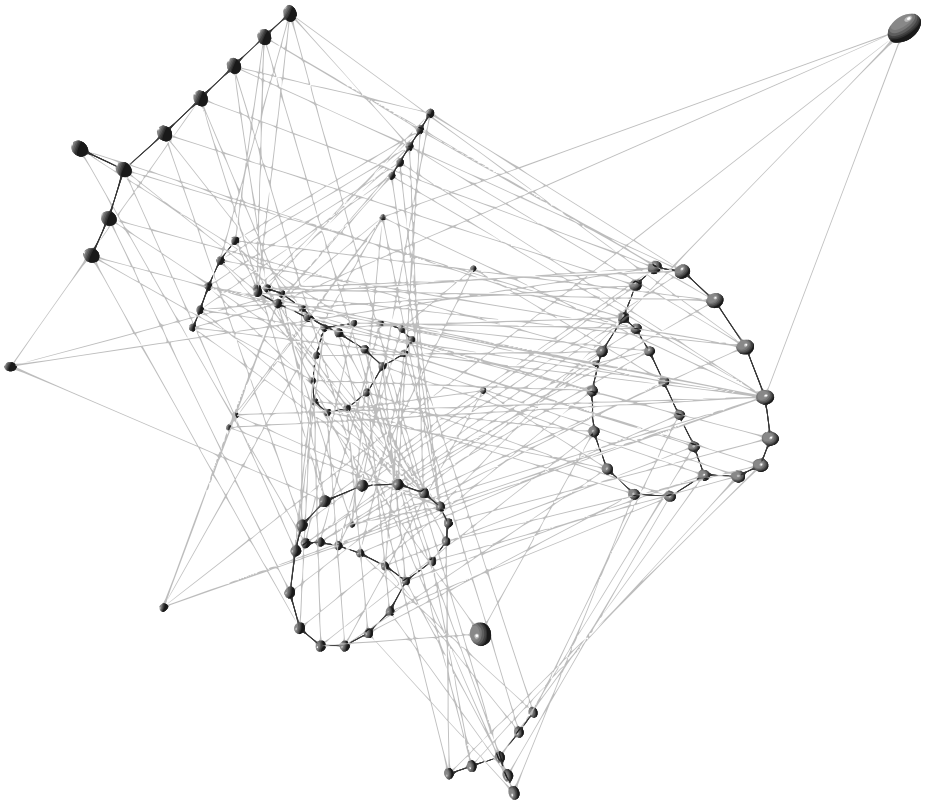


Figure 6.7: Graph #337 with apparent affinity webs. For more visibility, affinity edges are drawn in solid black lines and interferences in grayed lines.

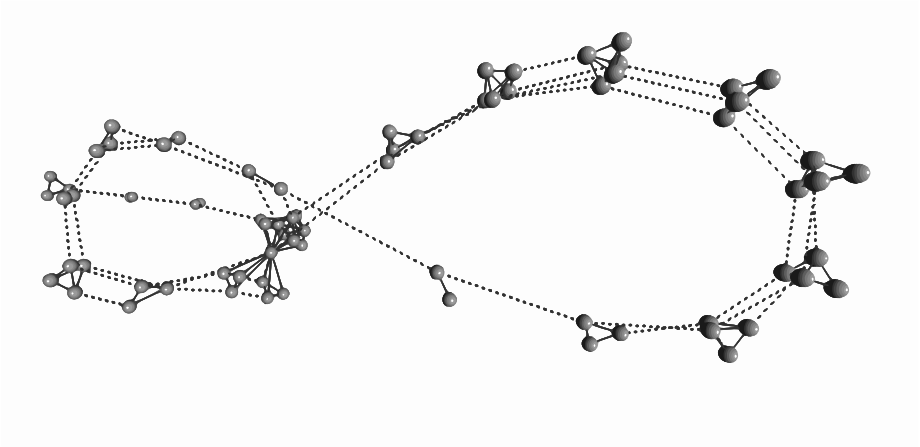


Figure 6.8: Graph #105 after applying the optimal "clique" rule. There remains cliques that "miss" one affinity to be coalesced by this rule.

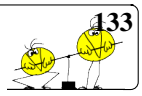
Indeed, in aggressive coalescing, there is no reason to coalesce x and y if they belong to different webs: there will be no gain in terms of affinities, and it could constrain further coalescings. Hence, before applying optimal rules, a graph can be partitioned into maximal connected component (affinity-wise), and independent work can be done on each different connected component. Hence only interferences between two nodes of the same web constrain the aggressive coalescing.

Moreover, the "terminal" rule has now the ability to *disconnect* connected components. This can indeed happen if the cost of the simplified affinity is larger than the cost of the affinity that remains. In that case, the new affinity has either cost zero (i.e., useless to coalesce) or a negative cost (better not to coalesce it). If this affinity is an isthmus (or "cut-edge," i.e., an edge that disconnects a component if it is removed), it will never be beneficial to coalesce such an affinity so it can be removed from the graph. Beware that if the affinity is *not* an isthmus, it is wrong to remove it from the graph even if it has a negative cost. Indeed, there could be a path of affinities that would still "want" to coalesce x and y, its two extremities. In this case, the information that by coalescing them one will "lose" something must be present: indeed, this means that it will not be possible to coalesce the simplified affinity too.

To conclude, the terminal rules did not work for conservative coalescing because the conditions were too strong. In the context of aggressive coalescing where clique and terminal rules are alternated, the separation of connected components in webs makes the conditions for terminal rules much more probable.

6.4.4 Disclaimer

We recently found a technical report by Blazy and Robillard [2008] presenting a rule very similar to our "clique" rule. They also use their rule in the context of the Coalescing Challenge, to optimize the ILP formulation of Grund and Hack [2007]. It is not very surprising that by looking at the shape of the graphs, they developed a similar rule, as it is visually clear that many affinities between cliques are unnecessary. They managed to treat one more case by remarking that, on some graphs, not all nodes of two consecutive cliques are linked by affinities in a one-to-one manner: sometimes, there is one



node on each consecutive clique that does not have any affinity, as shown on Figure 6.8. With our current rule, we will not allow to merge to such consecutive cliques, while it is still optimal. In fact, this can be done in our formulation: it is possible to consider that these nodes are linked by affinities of null weight, and then our clique rule would find them. This would require modifying Function `Clique_rule` at Line 15.

Another difference is that, in their algorithm, they restrict themselves to cliques while we included in our rule the possibility that the components we want to coalesce together are not cliques. In our formulation, the restriction is just that one component should be structurally a subgraph of the other, plus, they are allowed to have neighbors, provided they are common to the two components. This could be useful if the components of the split graphs were each constituted of a union of two cliques, as we would have expected the graphs to be, or to coalesce two cliques that have common pre-colored neighbors.

6.5 Experiments and evaluation

6.5.1 Methodology

Coalescing is challenging for graphs with many affinities and high register pressure. In particular, a graph-based spill-everywhere algorithm leads to a too simple coalescing problem. It this thus difficult to find a good set of benchmarks, hard enough to solve, large enough, to experiment and evaluate various strategies in detail. The Coalescing Challenge of Appel and George [2000] provides such an interesting collection of graphs on which state-of-the-art coalescing algorithms were tested. Also, Grund and Hack [2007] managed to give optimal solutions for all but three graphs using ILP, probably thanks to the low number of registers (six). We use them to compare the different heuristics, either optimistic or conservative. We point out that, to measure the quality of coalescing algorithms, it is more fair to work on such a collection of graphs than to measure execution time of codes for some platform: the latter gives results that are biased by many factors and are not reproducible, and anything can be claimed. Of course better execution time is the goal of coalescing, and here we measure the number (or weight) of coalesced affinities, i.e., of copies that are removed, which is always, in general, a benefit for the generated code. However, we will also see in Chapter 8, Section 8.2.2.2, preliminary experiments of how our algorithms perform under real conditions.

Benchmarks suite. In their spill algorithm, Appel and George performed live-range splitting at every program point. Hence the corresponding control flow graphs can be easily rebuilt. There are 474 graphs that correspond to regions (maybe procedures?) of the Standard ML of New Jersey benchmark suite, compiled for a Pentium with 6 general-purpose registers. On average, there are ≈ 26.7 basic blocks per region, with a maximum of ≈ 1090, and ≈ 231 instructions, with a maximum of ≈ 8300. Notice also that the architecture has many instructions with register constraints, which constrains the graph coloring—without necessarily simplifying it. This collection of graphs is thus interesting and representative.

Implementation. For our experimentations, we used a standalone program specially developed to coalesce Appel and George's graphs. It is implemented in Objective

Caml,[12] compiled with version 3.10.2. It consist of more that five thousand lines of code (and two thousand of comments). Experiments were done on an Intel Xeon running at 2.40GHz. Time measurements would probably be better if our algorithms were written in C, so the important time measure is the *relative* speed of execution between different algorithms.

Performance criterion. As the graphs correspond to live-ranges that are split at each program point, there are a large number of affinities in the graphs (compared to standard graphs where affinities usually correspond to original affinities, application binary interface (ABI) constraints, or possibly splits by SSA). Most of those are straightforward to coalesce, in particular, all those coalesced by the optimal "clique" rule. Therefore, it does not make sense to evaluate the quality of heuristics using a ratio with the total weight of initial affinities; it is more significant to focus on the affinities that are hard to coalesce, i.e., to use a ratio with the affinities that are *not* coalesced.

For that, to evaluate the quality of an heuristic h, we compute, for each graph, the cost $\hat{c}(h)$ of the coalescing given by h, i.e., the sum of the weights of the affinities *not* coalesced (remaining affinities):

$$\hat{c}(h) = \sum_{a \in Nc} w(a) \quad \text{where} \quad Nc = \{a = \langle x, y \rangle \in \mathcal{A} \mid a \text{ is not coalesced by } h\}$$

This cost in compared to the cost $\hat{c}(opt)$ of the optimal solution opt (provided by Grund and Hack).

$$q(h) = \frac{\hat{c}(opt)}{\hat{c}(h)} \qquad \hat{c}(opt) = \text{cost of optimal coalescing}$$

This gives us a performance ratio $q(h) \leq 1$ that measures the quality of a heuristic, i.e., the percentage (in weight) of the remaining affinities with h that could still be coalesced.

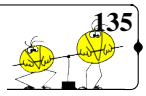
For example, if a heuristic h reaches 0.8, it means that, in the optimal solution opt, there are 20% (in weight) less affinities than for the heuristic h. The traditional performance ratio when evaluating algorithms is $\hat{c}(h)/\hat{c}(opt)$. The inverse, q, gives us a quick way to compare two heuristics h_1 and h_2 using $q(h_1) - q(h_2)$. For instance, if $q(h_1) - q(h_2) = 0.1$, we say that h_1 improves h_2 by 10%.

> Note: actually, to get an exact percentage when $q(h_1) \geq q(h_2)$, we should compute $1 - \hat{c}(h_1)/\hat{c}(h_2)$, but $1 - \hat{c}(h_1)/\hat{c}(h_2) = 1 - q(h_2)/q(h_1) = (q(h_1) - q(h_2))/q(h_1) \geq q(h_1) - q(h_2)$ and $1 - \hat{c}(h_1)/\hat{c}(h_2) \approx q(h_1) - q(h_2)$ when $q(h_1) \approx 1$. Thus, $q(h_1) - q(h_2)$ is a conservative estimation, h_1 actually improves h_2 by a bit more than 10% in our example.

Figures 6.9 and 6.10 give the average value of $q(h)$ for each heuristic. We also give a "weighted" ratio where graphs are weighted by their number of instructions so that bigger graphs (such as #139), usually more difficult to coalesce, get more importance than very small ones (such as #098). As we will see, both average ratios lead to the same conclusion, i.e., if h_1 improves h_2 with one ratio, it also improves it with the other ratio. For this reason, and because we believe the weighted ratio is a better indicator of the quality of a heuristic, we refer to the weighted version when giving any numbers (percentages) in the discussions.

Note on affinity ordering. The order in which affinities are considered for coalescing is crucial, and it is a good idea, though not optimal, to consider them in decreasing order

[12] http://caml.inria.fr/ocaml



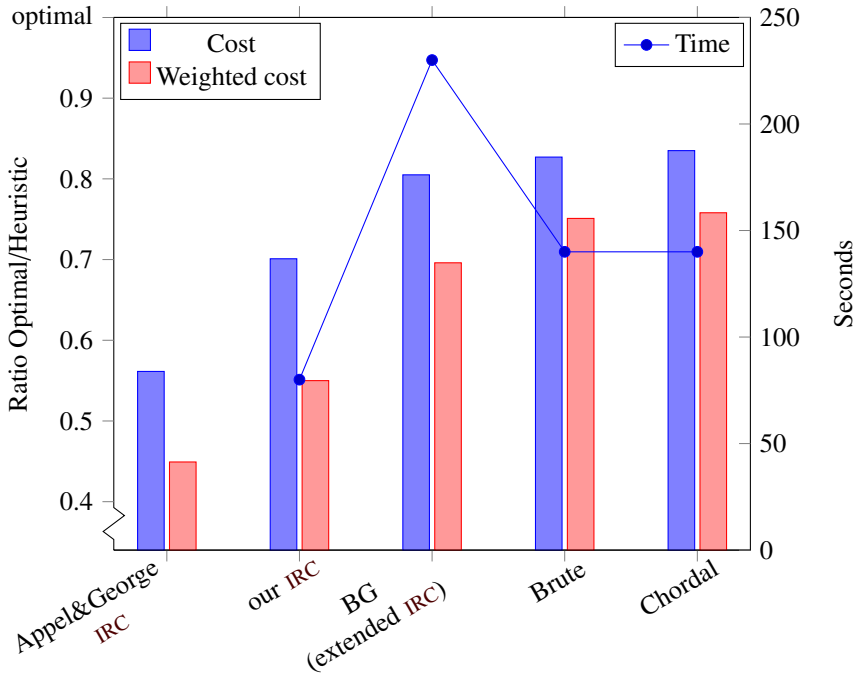


Figure 6.9: Comparison of conservative heuristics.

of their weight, so that bigger affinities get coalesced first. However, many affinities have the same weight, and some total ordering must be chosen. This is never tackled in the coalescing literature, although it is a known fact that tie breaking is important when solving heuristically NP-complete problems. For instance, Gibbons and Muchnick [1986] spend much of their article on how to choose instruction scheduling when given the choice, and Briggs et al. [1994] remarked that simple things like node ordering and how ties are broken lead to sometime "anomalous" behaviour they call "NP-noise." As we will see, the ordering choice for coalescing has a strong impact on the quality of the results.

It seems reasonable to guide the ordering using, for example, the knowledge of the program structure or some graph properties (e.g., node degrees). We will discuss several orderings in Section 6.5.4. Before, to provide reproducible results, we chose a deterministic ordering: in case of equal weight, we use the order in which affinities appear in the graph description file ("first seen, first taken"). This ordering is not arbitrary since it exactly follows the CFG of the original program. Notice that using another ordering does *not* change the overall *relative* comparisons of the different schemes, except for the external results—given by Appel and George [2000] on the Coalescing Challenge web page—for which we do not know the ordering that was used. Nevertheless, we also give the results for our implementation of these schemes.

6.5.2 Conservative heuristics

Figure 6.9 shows the quality of the conservative heuristics for the criterion $q(h)$. The optimal has value 1, hence the higher a heuristic, the better. Each heuristic is evaluated

by the average on all graphs—left column, blue and labeled "Cost"—, the weighted average on all graphs—right column, red and labeled "Weighted cost"—, and, when available, i.e., for our implementations, the overall time (middle bar, yellow and labeled "Time") spent by the heuristic on all 474 graphs.

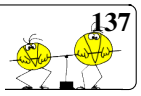
The first results are the performance of Iterated Register Coalescing as given on the Coalescing Challenge web page [Appel and George, 2000]. The second heuristic is *our* implementation of IRC, which gives 10% better results, although it is the same algorithm. Note that it may be due to the use of a different affinities ordering, an information that was not provided by the authors. The third heuristic, `BG`, uses the rules of Briggs and of George, extended to any type of nodes (pre-colored or not) as explained in Section 6.2. This simple change improves by 15% the quality of our IRC implementation, but at the price of a (roughly) 3× slowdown. Finally, the last heuristics, `Brute` and `Chordal`, implement both Function `Brute_Force_Improved` (page 108), with an additional call to Function `Chordal_Coalescing` (page 114) for the second, as described in Section 6.2. The `Brute` heuristic improves `BG` by 5%, hence it is 20% better that standard IRC—30% if compared to the results provided by Appel and George [2000]—while being only 1.7× slower.

For our developments, we started with an implementation of IRC and experienced that the effort to extend it into an implementation of `Brute` was small, making this improvement worthwhile. Also, we measured that, without the improvements proposed in Section 6.2.1, i.e., with a naïve use of Function `Is_kGreedy` (page 104) for each tested affinity, the heuristic `Brute` processes all graphs in more than one hour instead of 135 seconds: it is actually 10× slower on average and more than 30× slower on the biggest graphs, which are responsible for about three quarters of the time spent. Finally, the chordal rule added in `Chordal` only improves `Brute` by around 1%, while being more complicated to implement. However, the execution time overhead is not significant, so it is a "free" percent for whoever needs it and is ready to implement it.

6.5.3 Optimistic heuristics

Figure 6.10 shows the quality of the aggressive heuristics. The first heuristic is the variant of Park&Moon optimistic coalescing developed by Appel and George [2001] and whose results are, again, provided on the Coalescing Challenge web page. The next three heuristics are our (unsuccessful) attempts to reproduce these results: these are the heuristics 1, 2 and 3 of Section 6.3, i.e., Appel&George type, Park&Moon type, and our "iterated" version of Park&Moon. Strangely, they give results worse by 10% to 25%. The fifth heuristic, `De-coalescing`, uses our de-coalescing scheme, after an aggressive part, as explained in Section 6.3. It alone gives results of the same quality as the optimistic provided by Appel and George, but requires more time than our implementation of optimistic coalescing. However, our scheme produces a greedy-k-colorable graph and was designed to enable a conservative coalescing post-pass. So, we tried, after `De-coalescing`, two conservative techniques, the cheapest and less aggressive one, i.e., Briggs's and George's rules (`BG`), and the most aggressive one, our chordal rule (`Chordal`). These rules use a little more time and improve the results by 2.5%. This is not much compared to Appel&George version of optimistic coalescing (first column), but it is 13% better than our implementations of optimistic coalescing, with a 2× slowdown. According to these results, it appears that, after our de-coalescing, Briggs's and George's rules are enough to eliminate many useless de-coalescings.

Compared to the conservative heuristics, the best optimistic coalescing scheme equals the best conservative one (the unweighted average is 2% better with optimistic



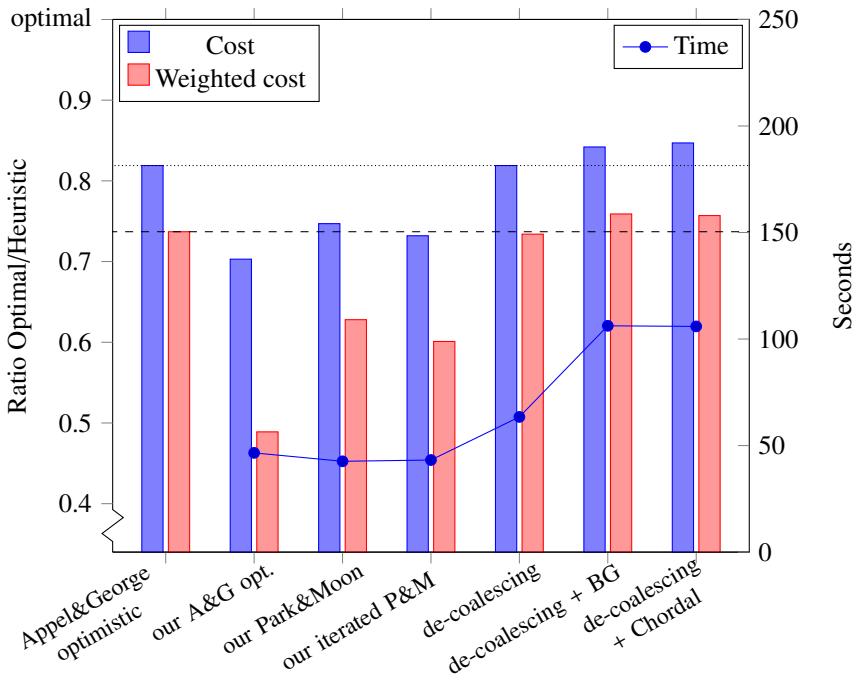


Figure 6.10: Comparison of aggressive heuristics.

coalescing) and it is about 30% faster. Also, the results are just slightly better than Appel&George version of optimistic coalescing. However, the next section shows that far better results can be obtained, especially with conservative coalescing, thanks to different (so, better) affinities orderings.

6.5.4 Ordering the affinities

In Section 6.5.1, we mentioned that the ordering of affinities of equal weight has a strong impact on the quality of the results. The results of Sections 6.5.2 and 6.5.3 correspond to a particular canonical ordering (order of the program, basically). We now show that, with an adequate ordering of the affinities, our algorithms can perform better. All the orderings we tried consider affinities by decreasing order of weights, since it is usually better, though of course not optimal, to first coalesce affinities that cost more. The tie-breakers we tried in case of equal weights are the following:

1. *Program*: the affinity appearing *first* in the program, i.e., in the graph description file, gets the priority.

2. *Reverse*: the affinity appearing *last* in the program, i.e., in the graph description file, gets the priority.

3. *Lexico*: first, let us define an ordering on the nodes. For initial nodes, $x < y$ iff x appears before y in the graph description file—i.e., the identifier of x in this file (an integer) is smaller than the identifier of y. For a coalesced node x, its identifier is set to be the one of the smallest node coalesced with x. In that case,

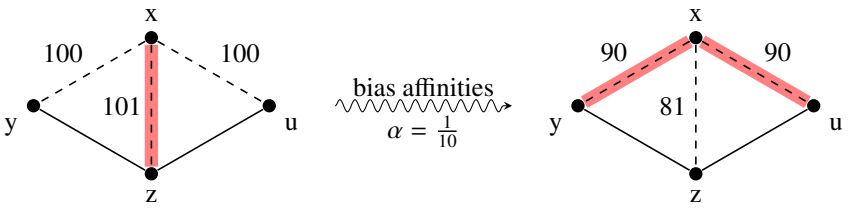


Figure 6.11: Motivation for biased affinity weights.

let x_s and y_s be the smallest of the nodes coalesced respectively with x and y, then $x < y$ iff $x_s < y_s$.

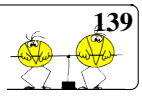
> Note: this defines a total ordering since every node has initially different identifier in the graph description file, which is an integer hence < defines a total ordering. Then, any coalesced node has a unique identifier since a node cannot be coalesced with two different nodes.

The affinity $\langle x, y\rangle$ then gets priority over $\langle x', y'\rangle$ iff $(x, y) < (x', y')$ using the lexicographic ordering based on the node ordering, i.e., iff $x < x'$ or $x = x'$ and $y < y'$.

While using custom ordering, it is important to update carefully the affinities when the graph changes to keep the benefit of using a good ordering. If $\langle x, z\rangle$ and $\langle y, z\rangle$ are two affinities, and x is merged with y to form the node xy, the two affinities are replaced by $\langle xy, z\rangle$ with weight $w\langle x, z\rangle + w\langle y, z\rangle$. In that case, the ordering must be updated: for *Program* (resp. *Reverse*), the order of $\langle xy, z\rangle$ is the minimum (resp. maximum) order of $\langle x, z\rangle$ and $\langle y, z\rangle$; and *Lexico* already defines the behavior for coalesced nodes.

Biased affinity weights. We also modified the global ordering of affinities because of the following remark. Suppose $\langle x, y\rangle$ and $\langle x, z\rangle$ are two affinities such that y and z interfere (see Figure 6.11). Coalescing both affinities is not possible as coalescing one constrains the other. When coalescing $\langle x, y\rangle$, $w\langle x, y\rangle$ is saved and $w\langle x, z\rangle$ is lost. This becomes a problem if there is another affinity $\langle x, u\rangle$ where z and u interfere. If $\langle x, z\rangle$ has a weight even slightly greater (say 101 versus 100), it will be chosen first, and this will prevent coalescing the two others. Here, the final cost will be 200 while choosing to coalesce $\langle x, y\rangle$ and $\langle x, u\rangle$ leaves a cost of 101. To avoid this situation, we devised a strategy called "*bias*." When applied, our algorithm works with modified weights, computed from the initial weights. Of course, the final cost of the remaining affinities is still computed using the initial weights. For each affinity a, we initialize $w_{bias}(a)$ to $w(a)$. Then, whenever there is a triangle x, y, z such as in Figure 6.11—i.e., where y and z interfere and the two affinities $\langle x, y\rangle$ and $\langle x, z\rangle$ exist—we subtract $\alpha \cdot w(x, z)$ to $w_{bias}\langle x, y\rangle$ and $\alpha \cdot w\langle x, y\rangle$ to $w_{bias}\langle x, z\rangle$. We fixed $\alpha = \frac{1}{10}$, arbitrarily, which gives the desired behavior.

Results. Figure 6.12 shows the results of our best optimistic and conservative algorithms with different affinities orderings. The first two columns are two optimistic versions based on the algorithm of Park and Moon [2004]: the one provided by Appel and George, and the best we managed to reproduce (a "Park&Moon type" (see Section 6.3), using the *Lexico* ordering and *bias*), which is still not as good. The next



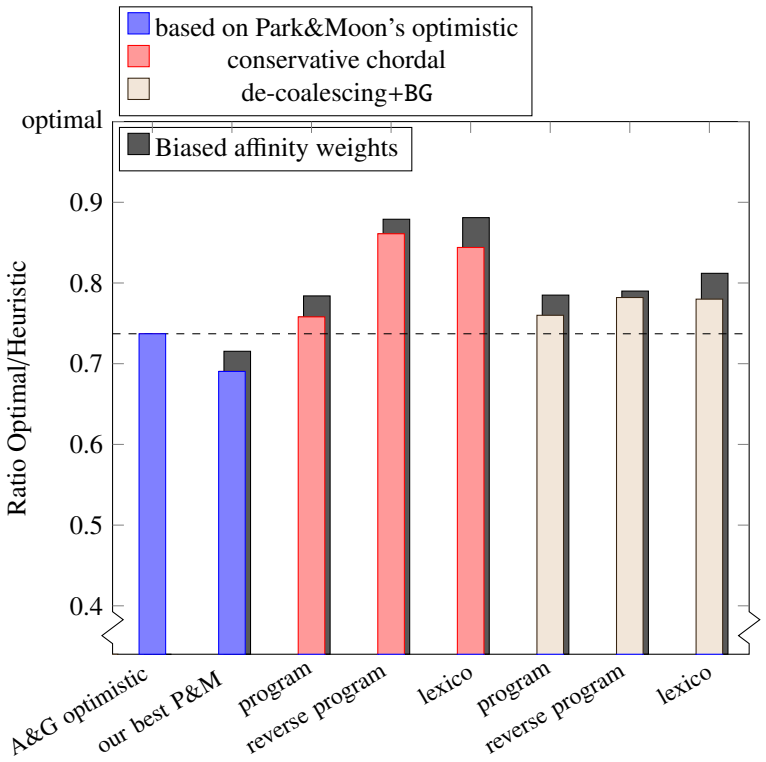


Figure 6.12: Quality results for different affinity orderings.

three columns show the conservative results. We displayed only results for `Chordal`, which performs best, but the effects of ordering are similar on the other conservative techniques (in particular `Brute`, which has equivalent results). Here, using the *Reverse* order instead of the normal program order greatly improves (by 10%) the quality. When using *bias*, both *Lexico* and *Reverse* orderings give the best overall results. This means that the optimal solution can improve the result only by 12%, and this is more than 15% better that Appel and George's version of optimistic coalescing, but for a 3× slow-down. Finally, the last three columns show the effects of ordering on our best optimistic technique, `De-coalescing` followed by `BG`. So far, we point out that it is not clear which ordering works best (in particular *Reverse*) and why. This is an important open question.

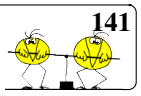
To conclude, for the optimistic strategies based on Park and Moon's algorithm and for the optimistic strategies based on our de-coalescing technique, the affinity ordering plays a role, but definitively not as important as for our conservative techniques. The same holds for *bias*. Our interpretation is that the aggressive part of optimistic-like schemes may coalesce the wrong affinities, and since it is aggressive, it will coalesce them whatever the ordering is. The current de-coalescing phase is then unable to decide to de-coalesce the "bad" nodes created. To confirm these doubts, we ran the following simple experiments: on the one side, `Brute` only, and on the other side, `Brute` followed by aggressive coalescing, then de-coalescing, then `Brute` again. The second strategy is more that 5% worse than the first one. This means that, among the affinities not coalesced by `Brute`, the aggressive part chooses to coalesce some that are not de-coalesced later: instead, some nodes created by merges *in the first phase* of `Brute` are de-coalesced. This shows it is quite difficult in an optimistic strategy to undo properly the bad effects of aggressive coalescing, i.e., the fact that it does not take colorability into account.

6.5.5 Using the optimal "clique" and "terminal" rules

6.5.5.1 Use of the "clique" rule

As explained in Section 6.4.1, the "clique" rule is able to coalesce affinities of trivially unnecessary split points in the program. By applying this rule (in its conservative version) repeatedly on a graph from the Coalescing Challenge, one gets an idea of how many split points were interesting for coalescing in the program. We did it on all the 474 graphs, and compared the number of nodes and affinities in the resulting graphs to these numbers in the original graphs. Figure 6.13 shows the distribution of the ratios for affinities and nodes using a box plot. For most of the graphs, there is about three to four times fewer nodes and between three to five times fewer affinities after applying the "clique" rule. This means that no more than about one third of every possible split point in a program is potentially interesting for inserting shuffle code. This is still a lot, and makes us think that SSA split points, which are in a far fewer number, are probably not enough for a coloring heuristic that needs to insert permutations of colors at split points to avoid spilling. Better split points might be found inside basic blocks, and not just on the incoming edges of basic blocks.

What are the effect of using the clique rule during coalescing? There are two kinds of improvements, both for the incremental conservative coalescing approaches. First, the clique rule improves the speed of coalescing. For the `Chordal` scheme, we managed to lower the time for coalescing all graphs from 135 seconds down to only 57 seconds, making it more competitive in time to optimistic-like strategies. Indeed, the



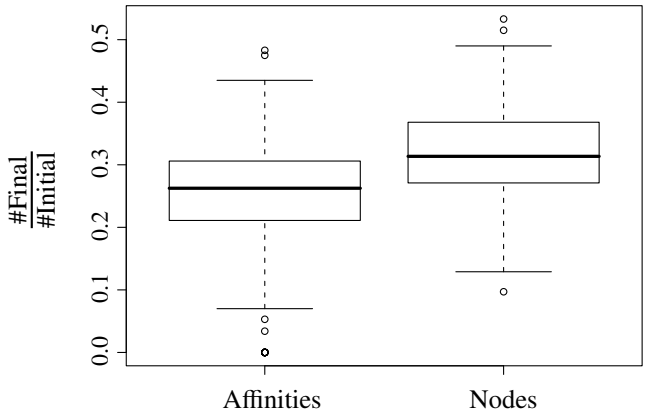


Figure 6.13: Graph size reduction after using "clique" rule: the y-axis is the ratio between the final and the initial number of affinities or nodes.

aggressive pre-pass of such strategies greatly speed up the process, while incremental-like strategies suffer from a really high number of affinities. The clique rule manages to find rapidly sets of affinities to coalesce, more quickly than the local rules of Briggs and George since, at about the same price, it can coalesce more than one affinity at a time.[13] Moreover, if the clique rule is used to create a smaller graph description file beforehand, the memory print of the interference graph is much smaller when doing coalescing, which reduces significantly the processing time for very big graphs. For instance, it takes 14 seconds to coalesce #139 using the clique rule followed by the `Chordal` scheme, and only 10 seconds if using the clique rule, writing to a new file, then applying `Chordal` to the new (smaller) graph (1.4× faster).

Second, the clique rule improves the quality of the resulting coalescing: the best `Chordal` scheme on the orderings tried is now 2% better than the previous one. This is a good news and means that our brute force algorithm will not find necessarily find the "unnecessary" splitting points. On worse orderings, the clique rule can improve by up to 7% the results of the `Chordal` scheme. Thus, it seems that coalescing first affinities that are safe from a cost point of view reduces later errors when our heuristic tries affinities in a not-so-good order.

As for the optimistic approaches, the clique rule does not improve them as it did for the conservative ones. All affinities coalesced by the clique rules can also be coalesced by the aggressive phase, only faster since it does not have to perform the check on the weights of affinities. Hence using the clique rule makes our optimistic algorithm run more than 2× slower. We did not observe such a slowdown when creating new graph description files: in this latter case, the whole coalescing is faster than before, taking also about 55 seconds instead of the initial hundred of seconds. This makes us think that the de-coalescing phase of the former experiment probably takes a lot of time de-coalescing affinities coalesced by the clique rule (these can be cheaper than the others which are coalesced in decreasing order of weights), which is obviously useless. Concerning the quality of the coalescing, it does not help whenever the ordering was already the best. However, as in the conservative case, it also helps whenever the

[13] For an affinity $\langle x, y \rangle$ between two 6-cliques, Briggs and George check the degree of 12 nodes. The clique rule also traverses 12 nodes but in the end, manages to coalesce 6 affinities at a time, making it about 6 times more effective.

affinity ordering is not very good. This confirms our thought that the clique rule helps reducing the disadvantages of having a bad ordering of affinities with equal weight.

6.5.5.2 Use of optimal rules in aggressive coalescing

In the context of conservative coalescing, the "terminal" and "double terminal" rules were never used. This is not a big surprise since it would require one variable (the terminal node) to be simultaneously alive with at most only one other variable. But originally, we thought of this rule while working on aggressive coalescing, in which case it is possible to separate the graph in "webs," as explained in Section 6.4.3, and then candidates to be terminal nodes must have at most one neighbor *to which there exists a path of affinities* (because they are in the same connected component in terms of affinities). In this aggressive context, the terminal rules are used and the following table shows the statistics:

263	coalesced by "double terminal"
1,230	simplified by "terminal"
954	simplified and coalesced by "terminal"
91	coalesced by "clique" thanks to "terminal" rules
$3.57 \cdot 10^5$	coalesced by "clique"

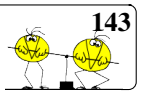
As expected—because the requirements are stronger—the "double terminal" rule is less used that the "terminal" rule. Since 1,230 affinities were simplified by the "terminal" rule, 2,460 affinities were in fact concerned (counting the affinities that remains in the graph), and in about 3/4 of the cases, the simplified affinity is the one that is coalesced in the end. But keep in mind that the bias was set towards such results since, with affinities of equal weight, the remaining affinity has cost zero and hence has few chances of being coalesced (the only possibility being that another path of affinities is completely coalesced). The last line serves as a comparison and shows that the "clique" rule is much more used than the terminal rules. Another experiment shows on the previous line that using the terminal rules "activates" some more possibilities for the clique rule, however, they are in very few number.

In terms of quality, we do not have the optimal solutions available for comparison. We present the total cost of the affinities not coalesced summed over all 474 graphs on the table below. One can see that separating affinities in webs is important for aggressive coalescing, but, as expected, the terminal rules are not very useful. The naïve strategy is the one used in our optimistic strategies. It coalesces affinities one by one as long as the two extremities do not interfere.

	Cost	Aggressive strategy
100.0%	30717277	Naïve
97.0%	29793964	Clique + naïve
90.4%	27769320	Clique with webs + naïve
90.35%	27754404	Clique and terminals with webs + naïve

6.5.6 Quality conclusion of the experiments

In conclusion, without a better ordering for aggressive coalescing, or a better de-coalescing part, our optimistic techniques do not reach the quality of our conservative heuristics, `Brute` or `Chordal`. They indeed work better than the other techniques, in other words, the non-conservative decisions taken by aggressive coalescing are hard



to repair in the de-coalescing phase, compared to an advanced conservative approach. In past studies, optimistic coalescing appeared better than conservative coalescing because Briggs's and George's tests are far too conservative, even in an iterated framework. These are not the results we were expecting. We used to believe that an aggressive strategy had better chances, as examples like the "diamond" graph of Figure 5.6 in the previous chapter let us think. But now, after working on both conservative and optimistic strategies, we think that it is much easier to guarantee that coalescing an affinity does not make a graph non greedy-k-colorable, than to find good indicators that a particular de-coalescing will help a graph to become greedy-k-colorable again. Indeed, in the latter case, it is hard to know where and why the colorability property was lost, while in the former case, one starts with a graph that already has the property we are interested in.

6.5.7 Inside the `Chordal` scheme

We end our experimental study with a deeper analysis of our most advanced conservative technique, `Chordal`, with a pre-phase using the "clique" rule, analyzing which rule decided to coalesce the affinities and the time it needed to take the decision.

In this scheme, affinities are first coalesced optimally (with regards to the final cost) by the clique rule, then, affinities are coalesced either by the `BG` rule, by the `Brute` rule if `BG` fails, or by the final `Chordal` rule if `Brute` fails. To evaluate the quality of these different rules, we implemented an "exact" conservative test using ILP: if the "path search" of `Chordal` fails on the remaining graph simplified by `Brute`, the low number of nodes (usually between 10 and 40) allowed us to check, exactly, if there exists a coloring of this remaining graph such that the affinity being tested can be coalesced. If such a coloring exists, the two extremities of the affinity and every other node with the same color are merged in the graph, provided that the resulting graph is still greedy-k-colorable.[14]

Compared to a pure `Chordal`, this strategy changes the affinities that are finally coalesced, but can still give an interesting view of the relative performance of each coalescing rule. The double pie chart in Figure 6.14 analyzes which rule was activated. The inner pie shows what happened to the 376,496 affinities of all 474 graphs. The clique rule first coalesced 71.5% of them, then the `BG` rules managed to coalesce 26%, and 0.5% were coalesced by `Brute` or `Chordal`, while the remaining 2% were not coalesced. This is the reason why `Chordal` and `Brute` remain fast. The outer pie classifies the 9406 (2.5%) affinities not coalesced by the clique or `BG` rules. Our heuristics achieve 19%: `Brute` coalesced 1524 affinities (16%) and `Chordal` 305 more (3%). Our ILP test found that 309 affinities (3%) could have been coalesced but only two third of them (197) led to a greedy-k-colorable graph. For 1127 affinities (12%), there was no coloring, and the remaining 6141 (66%) were constrained by other affinities and never tested. In comparison, the optimal solution (although optimized for weight and not number of affinities) leaves 5632 uncoalesced affinities (60%), which means there are 40% of "hard-to-coalesce" affinities. Of these, `Brute` + `Chordal` coalesce nearly half (19% to 40% is 48%), and (6141 + 1127 − 5632)/(9406 − 5632) = 43% of them *cannot be satisfied* even with an optimal incremental conservative technique, unless affinities are chosen in a different order. To conclude, this proves that `Brute` and

[14] If the graph is only k-colorable, merging all the nodes with the same color, for all colors, produces a k-clique, hence a greedy-k-colorable graph. But this would constrain too much the remaining graph and may prevent further coalescings.

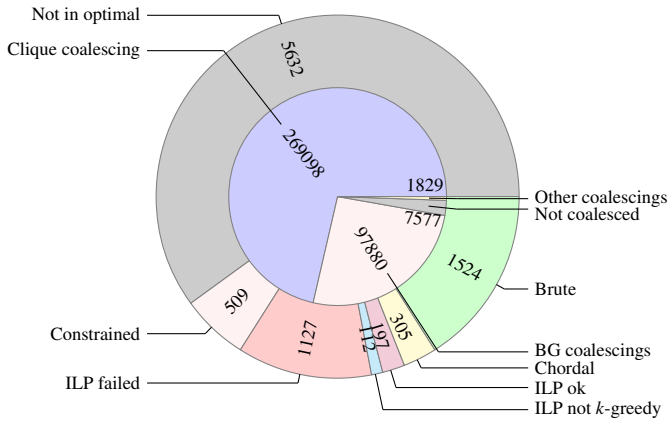


Figure 6.14: Effort of basic coalescing rules.

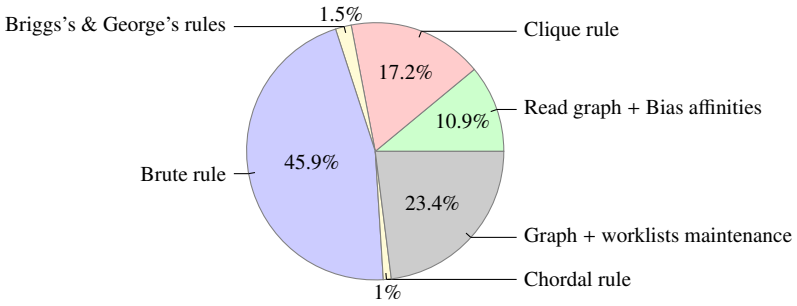
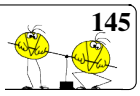


Figure 6.15: How the 57 seconds of `Chordal` were spent.

`Chordal` perform quite well since they coalesce 19% of all the affinities not coalesced by clique or `BG`, and an optimal incremental technique could coalesce at best 22%.

Surprisingly, for every affinity coalesced by `Chordal`, the remaining graph after simplification was an interval graph. Thus, `Chordal` applied positively only to the cases where it is optimal. Thus, using `Chordal` as a heuristic for greedy-k-colorable graphs does not seem to be efficient enough, at least for the graphs encountered here, to catch some of the remaining 3% available.

Figure 6.15 shows the time spent (a total of 57 seconds), for the whole set of graphs, in the different parts of the `Chordal` heuristic with a pre-pass of the "clique" rule. The `BG` rules and the overhead due to the chordal rule represent, respectively, only 1.6% and 1% of the time, much less than the 10.9% spent in initialization. The 23.4% used for maintenance concern graph and worklists updates (merging nodes, computing degree, sorting affinities, etc.), but does not take into account maintenance of worklists inside `Brute`. The latter is included in the biggest part of the pie-chart, the `Brute` rule, which takes 45.9% of the time. Finally, the clique rule, responsible for most of the coalescing, takes 17.2% of the time. This may seem a lot compared to `BG` rules, but this measure of the clique pre-phase includes graph maintenance which was technically difficult to separate. We also measured that, for small graphs, most of the time is spent in initialization, or in updating the graph and the worklists. As graphs grow bigger, the `Brute` part takes proportionally more and more time, as each test needs



to simplify nearly the whole graph. Actually, 27 of the 57 seconds were spent on the three biggest graphs, #99, #139 and #259, which contain respectively 20011, 28640 and 18445 nodes. We also point out that beyond the $2^{14} = 16,384$ nodes limit, our graph library cannot use an adjacency bit matrix anymore because of memory limitations. We need to rely on searches in adjacency lists instead. However, we measured that this is not the limiting factor.

6.6 Conclusion

It is a common belief that optimistic coalescing outperforms conservative coalescing. While trying to measure the quality of the conservative rules of Briggs and George, and based on our previous theoretical work from Chapter 5, we developed an advanced incremental conservative strategy that can coalesce more than half (in weight) of the affinities left by the well-known Iterated Register Coalescing (IRC) by George and Appel [1996]. It also outperforms the traditional optimistic coalescing of Park and Moon [2004] (adapted by George and Appel) by about 15%. Our conservative tests (brute-force or chordal-based) are more costly than the simple tests of Briggs and of George; however, by using them only when the quick tests fail and with additional implementation strategies, our final algorithm is less than 2× slower than the IRC of Appel and George, for the graphs of the Coalescing Challenge.[15] This is also because IRC may test some affinities several times while our heuristic tests each affinity only once and decides to coalesce or discard it. We found that using an optimal "clique" rule for conservative coalescing first (based on the particular shape of the graphs from the Coalescing Challenge) reduces the graph sizes and the overall time spent to coalesce in conservative strategies. We also developed a more aggressive optimistic approach that works for any greedy-R-colorable graph (as opposed to Appel and George's version of the optimistic algorithm) and ensures it is still greedy-R-colorable after coalescing. This enables the use of an additional (quick) phase of Briggs/George conservative coalescing, which leads to results better than traditional optimistic coalescing but still worse than our conservative strategy. We think this is because it is more difficult to devise a good de-coalescing indicator.

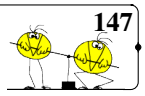
Very recently, Hack and Goos [2008] published a coalescing algorithm based on recoloring. We tried to include it in our tests, however, we did not find a mean to compare fairly our algorithms since their is included in their compiler and cannot be easily applied to graphs from the Coalescing Challenge. Conversely, the graphs they can generate from their compiler need to be modified so that they comply with the representation chosen by Appel and George. This constrains them more and makes the task harder for our external coalescing program.

During our experiments, we measured noticeable differences for the quality of the final results, depending on the ordering in which affinities with equal weights are considered. We verified that guiding this ordering by biasing the weights provides better results in average. However, we have difficulties explaining why some orderings are better than others. Hopefully, it seems that a pre-pass using the optimal clique rule lessen the final differences in quality. Our choice of using orderings based on lexicographic order or order in the graph description files was motivated by the fact that, in the literature, people do not usually have explicit orderings, making the comparisons

[15] With the clique rule as a pre-phase, the original IRC does not work because some pre-colored nodes need to be simplified for the graphs to be greedy-k-colorable. But compared to our "extended" IRC, we are still about 1.5× faster.

with their schemes difficult. Hence, we wanted to choose a total deterministic ordering on the affinities, so that our results could reproducible. We believe this point is worth exploring for both the aggressive/optimistic and conservative coalescing problems.

Finally, our linear-time incremental chordal-based conservative test for general greedy-R-colorable graphs provides only slight improvements over our "brute force" test. It is maybe a too direct adaptation of the optimal test for chordal graphs. However, we measured, thanks to an ILP formulation of the optimal solution, that our test misses very few of the remaining coalescable affinities. So, the real issue is now to find better orderings of the affinities.



Going uup!
Going doown.
(Laaby.)

Hyatt hotel elevator in Atlanta.

7

Parallel copy motion to get out of colored SSA

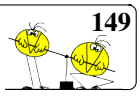
In the context of register allocation in two phases, the first phase handles the spilling, potentially with some splitting, so that Maxlive becomes less than R, the number of registers. Some more splitting is then done so that the interference graph gets easily colorable with R colors. The second phase then handles the coloring while trying to reduce the negative effects of the splitting of the first phase by performing coalescing. Different approaches of splitting exist for the first phase; for instance, aggressive splitting as done by Appel and George [2001], or splitting using Static Single Assignment (SSA). The first approach creates an enormous amount of new variables, which in turn makes the interference graph very big. This solution is not viable for just-in-time (JIT) compilation, where memory constraints are so tight that the cost building and storing the interference graph of the program—even in its non-split version—is often considered prohibitive. In this context, linear scan allocators introduced by Poletto and Sarkar [1999] are often preferred over graph-based register allocators because they do not need an interference graph.

SSA form is also becoming more popular in the context of JIT compilation since it still allows the compiler to perform important optimizations like common subexpression elimination or constant propagation without the drawbacks of keeping huge data structures in memory, or requiring a lot of computing power. For instance, [Wegman and Zadeck, 1991] use it for fast constant propagation, Boissinot et al. [2008] use the dominance property of SSA to calculate faster liveness, and it is also used by Sun's Java HotSpot™ client compiler [Kotzmann et al., 2008].

The problem with SSA is translating out of SSA. Indeed, ϕ-functions are not actual instructions (see Definition 2.25), and must be disposed of, as explained in Chapter 2, Section 2.3.6. Several alternatives exist, for instance Cytron et al. [1991] and Briggs et al. [1998] insert copies copies at the end of the preceding basic blocks. Sreedhar et al. [1999] do similarly, but they also add copies at the beginning of the block containing the ϕ-function. Leung and George [1999a]; Budimlić et al. [2002]; Rastello et al. [2004] and Hack et al. [2006] prefer to split critical edges to insert shuffle code.

Recently, solutions have been proposed to perform register assignment—assign variables to registers—while still under SSA, and then try to go out of *colored* SSA. This is the case of Hack et al. [2006] or Hack and Goos [2008] for instance. There are some advantages to this practice:

- copies are implicit: there is no need to add new copies and variables corresponding to a naïve out-of-SSA conversion;



- the dominance property can be exploited to perform a greedy coloring algorithm (using a "tree-scan" on the dominance tree for instance);
- one might use the SSA to perform other optimizations after coloring, like (cautious) code motion or scheduling;
- SSA has nice properties to avoid using an interference graph for coloring (liveness information is cheap, see Boissinot et al. [2008]).

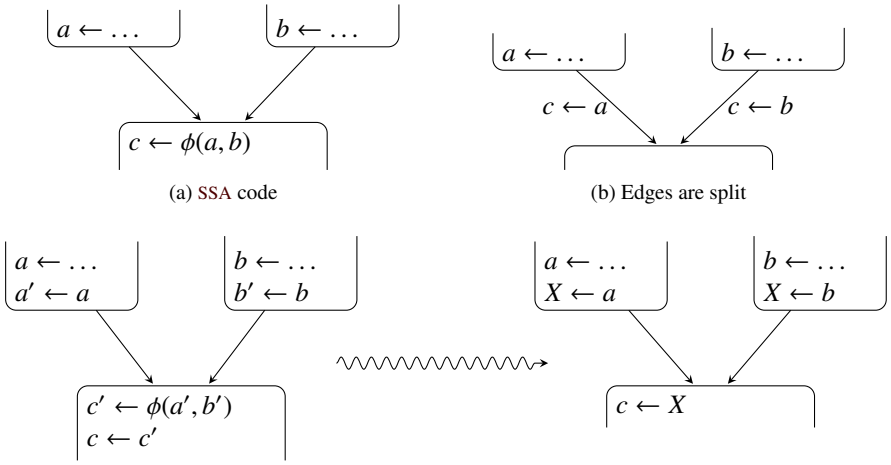
When going out of colored-SSA, it is easier to place the copies corresponding to ϕ-functions *on the incoming edges*. Indeed, we will see that it is not possible, at least in the general case, to create new variables and insert copies at the end of the preceding basic blocks or beginning of blocks. The problem with inserting copies on edges is that there is no basic block there. So, in order to actually add code, such an edge must be split and a new basic block must be created to hold the instructions. In this context, and supposing swap instructions are available, Pereira and Palsberg [2009] proposed an approach that does not require any extra register. However, there is a folk assumption that splitting edges is a bad idea. The main reasons for this are that:

- it adds one more instruction (a `jump`) (problem on highly executed edges); when splitting the back-edge of a loop, this prevents the use of a hardware loop accelerator;
- some compilers have abnormal edges that cannot be split (`gcc` or `open64` for instance, but it also depends on the target architecture);
- scheduling of instructions is not spawned across basic block boundaries. In particular, copies on a small basic block cannot be used to fill empty issues of bundles outside this basic block in Very Long Instruction Word (VLIW) architectures.

The goal of this chapter is to present a way to prevent the splitting of edges when going out of colored SSA, by moving the code that should be on edges to a more convenient place.

7.1 About going out of colored SSA

Performing the coloring of the variables under SSA, i.e., doing register allocation before translating out of SSA, fits our scheme of register allocation in two separate steps, first spilling then splitting, then coloring with coalescing. But then, going out of SSA is not classical anymore: one has to deal with actual registers already chosen to hold the values, instead of dealing with unboundedly many variables. This means that it is not possible to create new variables at convenience, as would do for instance Cytron and Gershbein [1993], Sreedhar et al. [1999] or Boissinot et al. [2009]. In the Sreedhar et al.'s algorithm, getting rid of ϕ-functions is done by adding copies of arguments in the preceding basic blocks. The trick is to insert another copy at the entry of the basic block of the ϕ-function, for the result of the ϕ-function. Then, all copies (of arguments and result) are renamed with a common name and the ϕ-function is not needed anymore. This scheme is presented with an example on Figure 7.1, where arguments a and b are put in copies a' and b', and the ϕ-function stores its result in c' instead of c. Then, all a', b' and c' are renamed with a common name, X.



(a) SSA code

(b) Edges are split

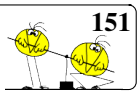
(c) Sreedhar et al.'s out of SSA

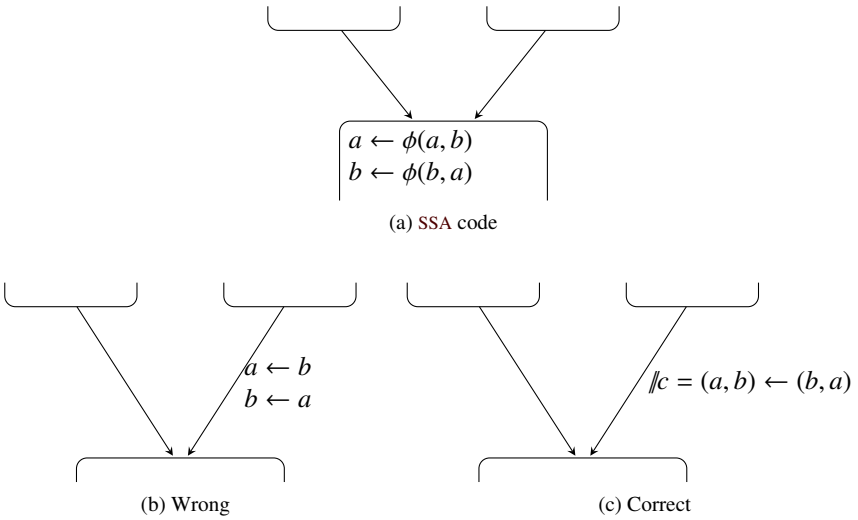
Figure 7.1: Going out of SSA: (a) original SSA code. (b) going out of SSA with the semantics of the ϕ-functions. (c) going out of SSA *à la* Shreedar.

Note: putting the result of the ϕ-function in a copy is important as is ensures that after renaming the copies, the variable will have a very small live-range. Cytron and Gershbein [1993] did not use it because they were still in conventional SSA, but Briggs et al. [1998] showed evidence that this method was incorrect after optimizations that break the conventional property, like value numbering. For instance, the result variable of a ϕ-function can interfere with one of its arguments, which can happen in a loop. Briggs et al. gave algorithms to solve this problem but they consider particular cases and need data-flow analysis. The solution of Sreedhar et al. [1999] is more generic and comes with a way to remove unnecessary copies, which makes it appealing to compiler writers.

Sreedhar et al.'s way of going out of SSA means that some extra registers might be required to hold copies of the variables. Indeed, it does not split on the edges as other translation out-of-SSA would do, meaning that the common variable X can interfere with other variables alive at the end of the predecessor blocks. In our colored case, it might be possible that all registers are already assigned to variables at this point. Whereas in Sreedhar et al.'s case, register allocation is performed later and they can still choose to spill some variables, we do not want to spill anymore, this is not an option for us, hence the right way of going out of colored SSA is to match closely the semantics of the ϕ-functions, i.e., to add copies on the incoming edges, as depicted on Figure 7.1b.

In this chapter, we deal with colored variables, and use the notation $x[R_i]$ to state that variable x is assigned to register R_i. In our case, the ϕ-functions represent a flow of values between registers instead of a flow of values between variables. For instance, $a[R_1] \leftarrow \phi(b[R_2], c[R_3])$ means that on the edge coming from the left, the value of R_2 must be transferred into R_1, while on the edge coming from the right, the value of R_3 must be transferred into R_1. Moreover, we have no way of creating "temporary variables" as we only have registers. Still, some registers might be free if the register pressure is lower than R. They can be temporarily used to keep some values.





(a) SSA code

(b) Wrong

(c) Correct

Figure 7.2: The swap problem: moves cannot be sequential or one of the variables is erased. Solutions can be to either use a temporary variable, or to represent the flow of values using a parallel copy.

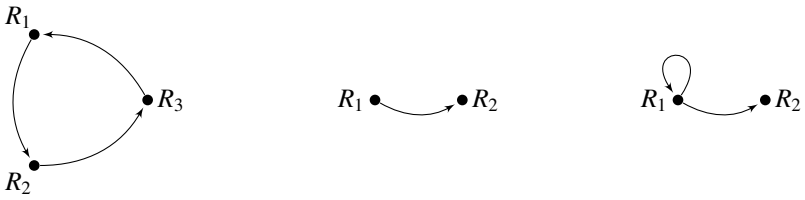
7.1.1 Introducing the parallel copies

As explained, special care should be taken when going out of colored SSA. A correct way to do it is to replace the ϕ-functions by *parallel copies* on the incoming edges. Parallel copies are virtual instructions that perform as many `move` instructions as required, all at the same time (see Definition 2.42).

The moves represent the flow of values that must be performed on these edges, and the parallel semantics is fundamental: when performing moves in a sequential way without any precaution, it is likely that a value will be erased before having a chance to be copied to its proper destination register. A classical example of this problem is known as the "swap problem," and described in Figure 7.2.

When dealing with colored variables, it is handy to implement parallel copies as arrays. Let $\|c$ be a parallel copy, then for each register $R_{i,\ 1\leq i\leq R}$, the i-th element of the array (indexed from 1 to R) indicates from which register the new value of R_i will be copied, i.e., if $\|c[i] = j$, the value of register R_j is copied into R_i during the parallel copy. A register that simply holds its value is represented as $\|c[i] = i$ and one that does not receive any live value is set to bottom: $\|c[i] = \perp$. Note that it is important to differentiate, in the parallel copy representation, registers that are *not* modified ($\|c[i] = i$) from register that do not receive any value ($\|c[i] = \perp$), since the latter are "free" registers and the former hold the value of live variables. When working with parallel copies, this allows one to quickly check which registers are free, hence usable as temporary value holders, and which are not. In this chapter we will also use a graphical representation of parallel copies, using a graph in which registers are nodes and directed edges represent the flow of the values. These are called "register transfer graphs" by Hack [2007] in his thesis and "windmills" by Rideau et al. [2008]. Figure 7.3 shows three examples of parallel copies.

For the same reason as before, self edges are important in the graph representation since parallel copies include *liveness* in their representation. In general, R_i holds the



(a) $/\!/c = [R_3, R_1, R_2]$ (b) $/\!/c = [\perp, R_1]$ (c) $/\!/c = [R_1, R_1]$

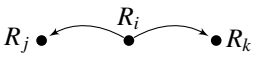
Figure 7.3: Examples of parallel copies, with their array and graph representations.

value of a live variable before the parallel copy if and only if (iff) there exists $1 \leq j \leq R$ such that $/\!/c[j] = i$, i.e., an edge leaving the node i in the graph representation. Register R_i holds the value of a live variable after the parallel copy iff $/\!/c[i] \neq \perp$, i.e., there exists an edge entering the node i in the graph representation. If $j = i$ the value stays in the same register, which is represented by a self edge. In this chapter, we will sometimes abusively refer to "live registers" when the actual rigorous meaning would in fact be "registers containing the value of a live variable."

Moreover, we consider that two registers containing the values of live variables at one point *interfere* at this point, even if the values are the same. Hence we forbid that a parallel copy defines a register more than once: two different registers cannot put their values into only one, i.e., there should not be two entering edges on any register on the graph representation of the parallel copy. This case can however happen, for instance if there were initially two ϕ-functions $a \leftarrow \phi(b, \ldots)$ and $c \leftarrow \phi(d, \ldots)$, but analysis determined that a and c do not interfere, and the register allocation allocated both a and c to the same register, say R_1. If b and d are in different registers, the parallel copy on the left incoming edge will try to put two values in R_1. But in that case, the program is correct only if b and d contain the same value. Hence only one of the two copies need to be performed, either $a[R_1] \leftarrow b[R_x]$ or $a[R_1] \leftarrow d[R_y]$. In that case, we arbitrarily choose one of them and we get back to our representation where parallel copies define at most one register. Hence for any register R_i, $/\!/c[i]$ is well defined.

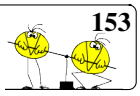
7.1.2 Duplications in parallel copies

Parallel copies can contain duplications. That is, the value in one register is copied into *two* registers (or more), as in the example Figure 7.3c. More formally, there is a duplication if for a register R_i every time there is two registers R_j and R_k ($j \neq k$) such that $/\!/c[j] = i$ and $/\!/c[k] = i$. Or, equivalently:



The value contained in register R_i is *duplicated*.As we will see later, duplications complicate the problem. They cannot be ignored as they are mandatory in many actual cases. For instance, if some variable a is used twice as argument in a ϕ-function to define two different variables. Since the two defined variables interfere,[1] a must be duplicated in another register before entering the basic block (see Figure 7.4a). Another

[1] Of course, this depends on the notion of interference, but we can safely suppose that some value numbering was performed to rename variables with the same value. Hence we can choose the "relaxed" Definition 2.8 of interference, in which the two results of the ϕ-function interfere since they are alive at the same time.



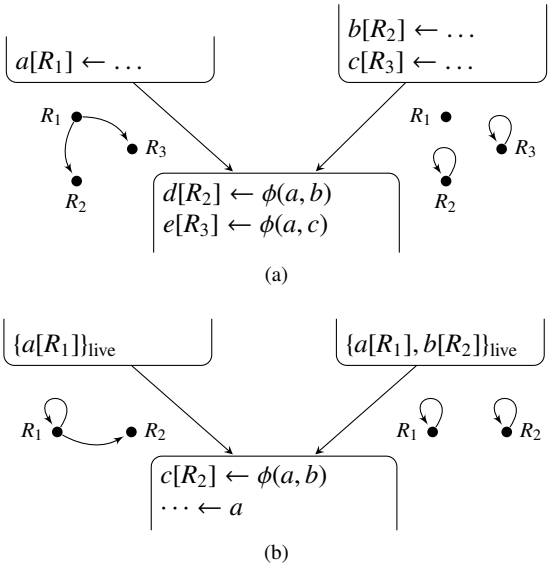


Figure 7.4: Examples of parallel copies where a must be duplicated: (a) because d and e interfere; (b) because a and c interfere.

example is given on Figure 7.4b: whenever a variable a is used in a ϕ-function, but is still live after it because it is used later.

One particularity of duplications is that the register pressure is higher after a parallel copy that contains duplications than it was before. This is one of the reasons why special care must be taken when dealing with them. We will see in the next section that another reason is that, in a parallel copy with duplication, the flow edges of values between registers cannot be "reversed" to obtain a parallel copy that has an effect opposite to that of the first one (else, there would be two edges pointing to the same register). For these reasons, we will try to get rid of duplications, which we will do in Section 7.3.1.

7.1.3 Reversible parallel copies

Parallel copies can be view as mathematical applications:

$$/\!/c : \{R_1, \ldots, R_R\} \to \{R_1, \ldots, R_R, \bot\}$$

If a parallel copy is injective, i.e., there is no two $i \neq j$ such that $/\!/c[i] = /\!/c[j]$, it is *reversible* in the mathematical sense. However, the $\bot$ value makes it difficult for parallel copies to be reversible. Indeed, it means that $/\!/c$ is not injective whenever more than one register does not receive the value of another register. Moreover, it makes the set of values different from the set of arguments, and we would instead prefer that $/\!/c^{-1}$ is also a parallel copy.

In fact, what the reverse of a parallel copy would mean? A natural way of thinking would be that $/\!/c'$ is the reverse of $/\!/c$ if the composition of the two is the identity, i.e., iff $/\!/c' \circ /\!/c = /\!/c \circ /\!/c' = Id$. But in fact we do not need the identity on the whole set of registers. What actually matters is that the registers that hold live variables beforehand

stay the same after applying $/\!\!/c$ and its reverse, i.e.,

$$\forall R_i \in \text{Live}, \quad /\!\!/c'[/\!\!/c[i]] = i$$

This means we can define $/\!\!/c^{-1}$, the reverse of $/\!\!/c$, as follows:

Definition 7.1. A parallel copy $/\!\!/c$ without duplication is called *reversible*. It reverse is the parallel copy $/\!\!/c^{-1}$ defined as follows:

$$\forall R_i, /\!\!/c^{-1}[i] = \begin{cases} j & \text{if } \exists j \mid /\!\!/c[j] = i \\ \bot & \text{otherwise} \end{cases}$$

Note that, since parallel copies contain liveness information, it is not possible to replace $/\!\!/c^{-1} \circ /\!\!/c$ by $/\!\!/c \circ /\!\!/c^{-1}$ in the general case. Indeed, *live_out* of $/\!\!/c$ is the same as *live_in* of $/\!\!/c^{-1}$ but is in general different than the *live_in* of $/\!\!/c$ (which in turn is the same as *live_out* of $/\!\!/c^{-1}$). However, in a context where Live = *live_in*($/\!\!/c^{-1}$), it is possible to insert $/\!\!/c \circ /\!\!/c^{-1}$ without breaking the code since it will perform the identity on the set of registers alive.

7.2 Properties for moving a parallel copy away from an edge

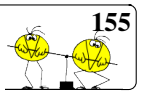
We have seen that, when going out of colored SSA, it is a good idea to place parallel copies on edges. But the goal of this chapter is to try to move these copies out of our way. In this part, we see that critical edges in the control-flow graph (CFG) make this task harder, and require the copies to be *compensated* on other edges. We consider a parallel copy $/\!\!/c$ on a edge E that cannot be split, or that we do not want to split. We see in this part what properties $/\!\!/c$ should have for us to be allowed to move it.

7.2.1 The problem of critical edges

When trying to move a parallel copy away from an edge E from source basic block B_s to destination basic block B_d, there are two possibilities: either move it *up*, i.e., to the bottom of B_s, or move it *down*, i.e., to the top of B_d, as explained by Figure 7.5a. Indeed, the code of the parallel copy needs to be performed whenever the execution path goes through the edge E. This works fine whenever E is the only edge leaving B_s or the only one entering B_d. However, if E is a *critical edge*, i.e., its source block has more that one successor blocks and its destination block has more than one predecessor blocks (see Figure 7.5b), it is in general wrong to move the parallel copy either on B_s or on B_d. Indeed, the code of the parallel copy should not be executed if an edge other than E is chosen (see Definition 2.40). A well-known example of the problem of critical edges is the "lost copy problem", depicted on Figure 7.6. In fact, in the case of a colored SSA code, simpler examples exist: it is sufficient that a parallel copy overwrite a register containing another live variable on the predecessor block.

7.2.2 Compensation

The idea to be able to move parallel copies away a critical edge E from B_s to B_d is still to move it to the bottom of B_s (resp. to the top of B_d), but to *compensate* its effects



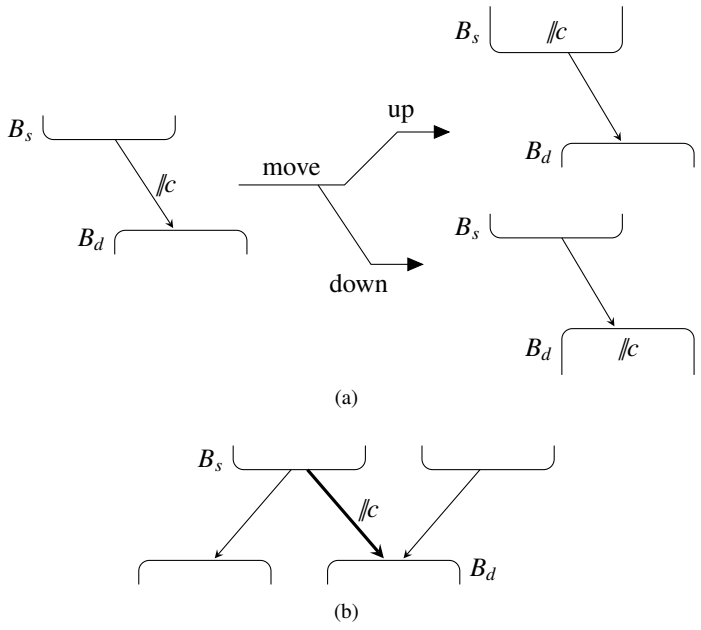


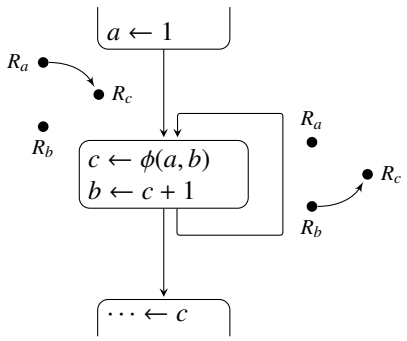
Figure 7.5: Moving a parallel copy. (a) if on a simple edge, it can be moved up or down. (b) if on a critical edge, this is not directly possible.

on the other edges leaving B_s (resp. entering B_d) by placing some code on them. This resemble the idea introduced by Fisher [1981] for trace scheduling, later called "compensation code," but it concerns general code and deals only with duplicating the code when moving instructions above a join point or below a split point. According to Freudenberger et al. [1994], it has been suggested that code could be inserted to undo any effects on "off-trace paths," but it is not done in practice because, even if it would be possible for simple register operations, it is too complex for general operations. We present in this section a way to "undo" the effects caused by parallel copies.

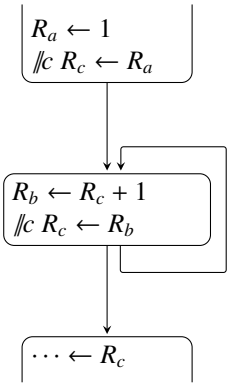
Figure 7.7 illustrates the notion of compensation, and Figure 7.8 gives three examples. These examples show three important facts: First, the notion of compensation appears strongly linked to a notion of "reversing" the effects of a parallel copy, Second, problems are different when moving up (post-compensation) or down (pre-compensation). Third, when moving up, the parallel copy changes to take the liveness of B_s into account.

Let us see in details the different problems:

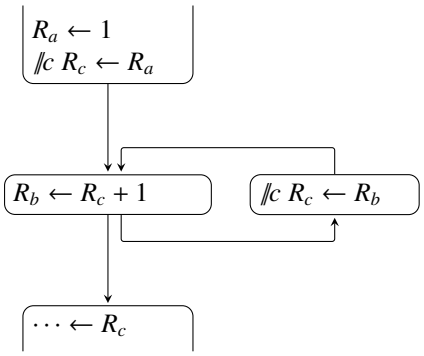
Move down. When moving a parallel copy down, one needs to pre-compensate the copy on other incoming edges. The idea is to "prepare" the registers so that, when arriving on B_d, $/\!/c$ will move the values into their right registers. Problems arise if $/\!/c$ contains duplications. Indeed, suppose that $/\!/c = [R_1, R_1]$ but R_1 and R_2 carry two different variables on another incoming edge. Then, whatever code is placed on this other edge, R_1 and R_2 will always have the save value after $/\!/c$ is executed at the beginning of B_d. There is no way to move a duplication down, since the semantics of the ϕ-function that produced this copy is: "if coming from the left edge, make the two registers equal, otherwise, make them different." This semantics is intrinsically linked



(a) Colored SSA code

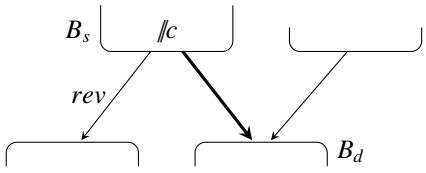


(b) Wrong

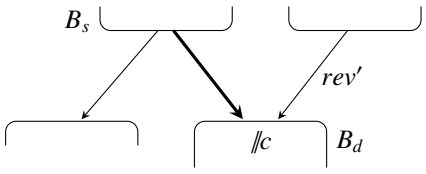


(c) Correct

Figure 7.6: Lost copy problem: (a) a,b and c reside in different registers; the back-edge of the loop is critical. (b) parallel copies are moved up (at the source of the edges), leading to incorrect value in c at the exit of the loop (one "+1" too much). (c) a correct way to go out of SSA is to move up the parallel copy at entry point, and to split the critical edge to place the other one.

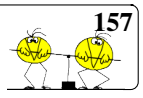


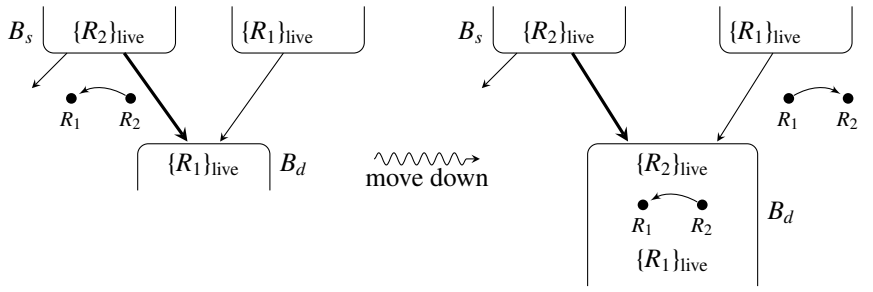
(a) Move up



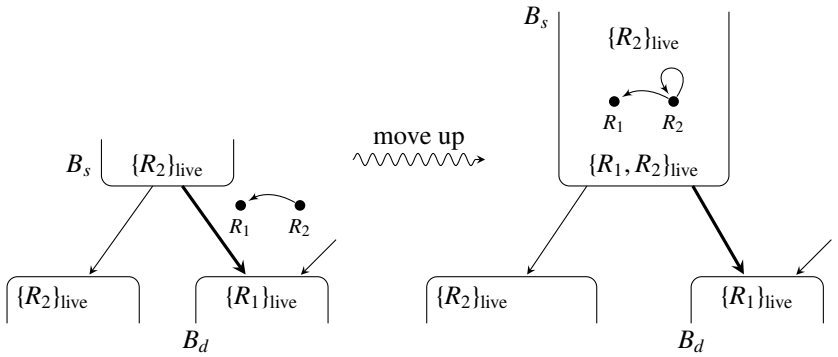
(b) Move down

Figure 7.7: Compensation: when moving a parallel copy out of a critical edge, compensation code *rev* must be added on other edges.

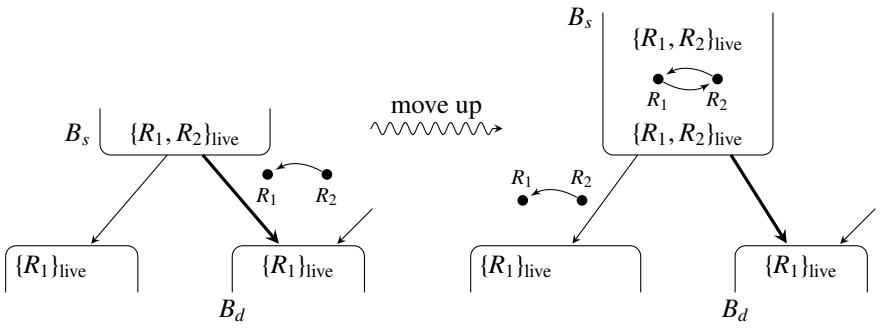




(a)



(b)



(c)

Figure 7.8: Examples of compensation when moving a parallel copy from a critical edge. (a) the compensation "prepares" R_2 with the value that must be placed in R_1 at the beginning of B_d. (b) R_1 is not live on B_s and can be overwritten; a self edge is added to R_2 to reflect it contains a live variable. (c) both R_1 and R_2 contain a live value on B_s. R_1 must be saved so registers are swapped. Then, R_1 must be restored with its original value on the left edge.

with an information on the flow of the program, information that is lost when entering B_d.

But then, what can be done if $/\!\!/_c$ contains duplications? We will see that duplications can in fact be moved up, and we will give in the next section a way to decompose a parallel copy so that the duplications are moved up and the remaining parallel copy is moved down.

Move up. When moving a parallel copy up, liveness problems arise. This was not a problem when moving down since *live_in* of destination basic blocks equals the liveness of their incoming edges. For source basic blocks, *live_out* equals the *union* of the liveness of their leaving edges. So, if $/\!\!/_c$ was created on the edge E from B_s to B_d, it carries the liveness information of E, which is a subset of *live_out*(B_s). That is why, in the examples of Figure 7.8, the parallel copy was modified when moved up and not when moved down. On (b), it needed to reflect that both R_1 and R_2 are *live_out* (since they are live-in of successor basic blocks); on (c), the two registers contain live variables, hence R_1 should not be overwritten. Its value is saved in R_2 using a swap instruction.

What about duplications? Duplications just make copies of registers. So, as long as there are enough free registers, it is possible to move duplications up. It is indeed not a problem to change the value of a register that will not be used on the other successor blocks. The only restriction is that duplications should not erase any live value. We will talk later of what to do if there are not enough registers.

We have seen with some examples that the compensation is some kind of *reverse*. Based on this idea, we will see in the next section how parallel copies can be decomposed into a part that contains duplications, and another part that is a reversible application, in the mathematical sense. This will allow us to develop a framework to move parallel copies away from edges.

7.3 Moving parallel copies away from critical edges

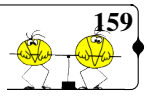
We have seen in the previous sections that parallel copies containing duplications are not reversible. But we have also seen that critical edges need compensation, and that compensation uses some kind of "reverse" of parallel copies. In this section, we will see how to move a parallel copy away from a critical edge. Figure 7.9 shows an example where a parallel copy that contains duplications is first de-composed so that the duplications are placed on the predecessor block, then it is moved down and compensated on the other incoming edge. There are two questions linked to the idea exposed on our example:

- How to decompose a parallel copy so that it gets reversible?
- How to move a reversible parallel copy?

We present in this section an algorithm to answer the first question, and then explain how to use *permutations* to solve the second question.

7.3.1 Decomposition of a parallel copy containing duplications

There are many possibilities of decomposing a parallel copy that contains duplications. Our interest here is to show that it is easy to find a working solution, and we propose an algorithm to compute it.



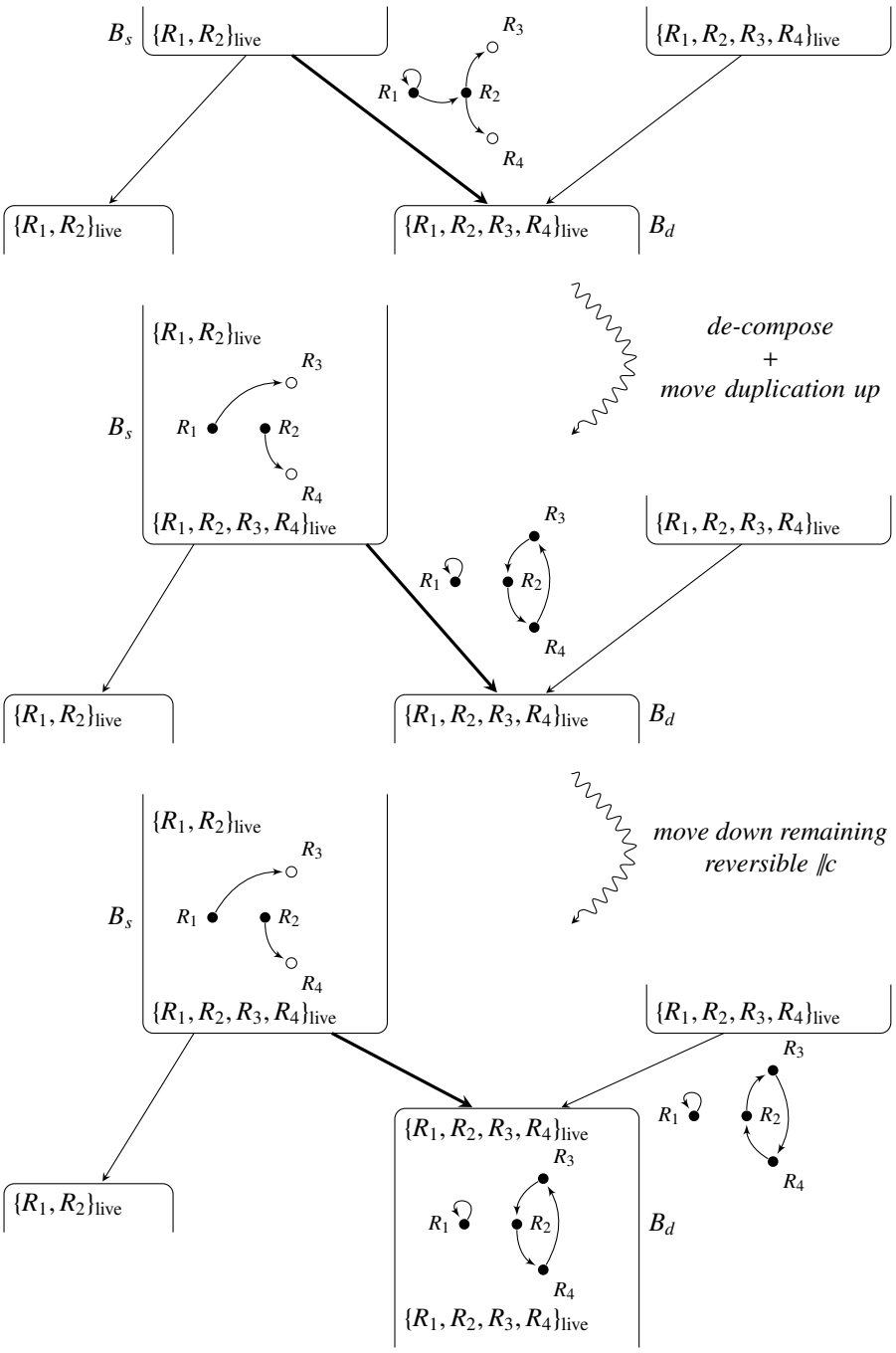


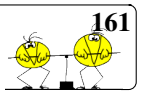
Figure 7.9: To move down a parallel copy with duplications: First, ∥c in decomposed into duplications followed by a reversible parallel copy, taking *live_out*(B_s) into account (free registers R_3 and R_4 are represented with empty bullets). The duplications are moved up in B_s. The remaining copy can then be moved down and pre-compensated on the other entering edge. Note that a swap of R_2 and R_3 also works for the remaining copy, instead of a cycle with R_2, R_3 and R_4 (but that would have been less didactic).

Whenever some duplications are found in a parallel copy $/\!/c$, it must be decomposed into $/\!/c_{f.rev} \circ /\!/c_{dup}$ where $/\!/c_{f.rev}$ is reversible and $/\!/c_{dup}$ contains the duplications. $/\!/c_{dup}$ will be placed at the bottom of the predecessor block, B_s, hence it should be constructed using informations on the liveness at the bottom of B_s. We propose the algorithm given in Function `De_compose`.If fact, in can be called directly on any parallel copy, even if the copy does not contain any duplication. It also expect as input B_s, the source basic block at the end of which duplications should be performed. This function adds instruction at the bottom of B_s (the duplications) and modifies $/\!/c$ so that it does not contain any duplication after the call.

Function `De_compose`($/\!/c, B_s$)

Data: Parallel copy $/\!/c$, basic block B_s where to put duplications.
Result: Duplications are added to the end of B_s and $/\!/c$ does not contain duplications anymore.

```
   /* Initialize data structures                                          */
 1 foreach i ∈ {1,...,R} do
 2     used[i] ← 0;
 3     targets[i] ← ∅ ;
 4 foreach i ∈ {1,...,R} do
 5     let j ← //c[i];
 6     if j ≠ ⊥ then
 7         used[j]++ ;
 8         targets[j] ← targets[j] ∪ {i};
 9 live ← live_out(B_s);
   /* Auxiliary functions                                                 */
10 Function is_Leaf(i): return ¬live[i] ∧ //c[i] ≠ ⊥;
11 Function is_Free(i): return ¬live[i] ∧ //c[i] = ⊥;
   /* De-compose //c                                                      */
12 foreach source ← 1to R do
13     while used[source] > 1 do
14         let target ← pop targets[source] ;
15         if target ≠ source. then          /* Look for a place to duplicate source */
16             if is_Leaf(target) then
17                 dest ← target ;
18             else
                   /* Try not to create cycles with target               */
19                 if ∃i, is_Leaf(i) ∧ ¬is_Reachable(//c, target, i) then
20                     dest ← i ;
21                 else if ∃i, is_Free(i) then   /* a free register avoids cycles */
22                     dest ← i ;
23                 else if ∃i, is_Leaf(i) then  /* choose in last resort a reachable leaf */
24                     dest ← i ;
25                 else
26                     Failure "Too many live variables!" ;  /* #live + #duplication > R */
27             generate instruction [R_dest ← R_source] at bottom of B_s ;
28             live[dest] ← TRUE;
29             //c[target] ← dest ;
30             used[source]−− ;
```

Function `is_Reachable`(∥c, *start*, *end*)

Data: Parallel copy ∥c, starting point *start*, end point *end*.
Output: TRUE if *end* can be reached from *start* in ∥c.

```
1  foreach i ∈ {1,...,R} do marked[i] ← FALSE;
2  current ← end;
3  marked[current] ← TRUE ;
4  while TRUE do
5      next ← ∥c[current] ;
6      if next = start then return TRUE;
7      else if next = ⊥ ∨ marked[next] then return FALSE;
8      else
9          current ← next;
10         marked[current] ← TRUE ;
```

We now explain our algorithm. Up to Line 9, the function just initializes some data structures. The array "used" contains how many times a register is used to define another one in ∥c, i.e., R_i is duplicated if used[i] ≥ 2. And the "targets" array remembers for each register the registers it defines in ∥c. We defined two particular types of registers, with functions to check quickly if registers are of one of these types on lines 10 and 11:

- the "free" registers: they do not contain the value of any live variable at the end of B_s (so they also do not on the edge of ∥c), and do not receive any value in ∥c;
- the "leaves:" registers that are leaves of the register flow graph, i.e., that receive a value but whose value is not used.

The decomposition algorithm checks for every register—variable *source* in the function—if it is duplicated (Line 13). In that case, copies must be added to B_s until only one register is defined using *source* in ∥c. If a duplication consists of a self edge, it does not need to be copied (but then, every other duplication involving *source* does). For a register *target* in which the value of *source* must be duplicated, the algorithm will first copy *source* into an available register *dest* at the end of B_s (i.e., *dest* does not hold a live variable). Then the copy from *dest* to *target* is added to ∥c. Four strategies are available to choose *dest*. They are illustrated in Figure 7.10.

1. take an immediate leaf (Line 16): the best possibility, then *target* = *source*, meaning the value is already put in its right register and will not need an additional copy later;
2. take a leaf that is not reachable by the target of the copy (Line 19): no cycle will be created in ∥c, and no additional color is required;
3. take a free register (Line 21): this means one color less but is better that creating cycles;
4. take a reachable leaf (Line 23): this will create a cycle in ∥c. Since there is no remaining free register, swaps (or a spill) must be used later if the parallel copy is then sequentialized at the bottom of B_s or the beginning of B_d.

If none of these strategies work, this means that there are too many duplications. Indeed, there must be at least as many registers *not alive* at the bottom of B_s as the

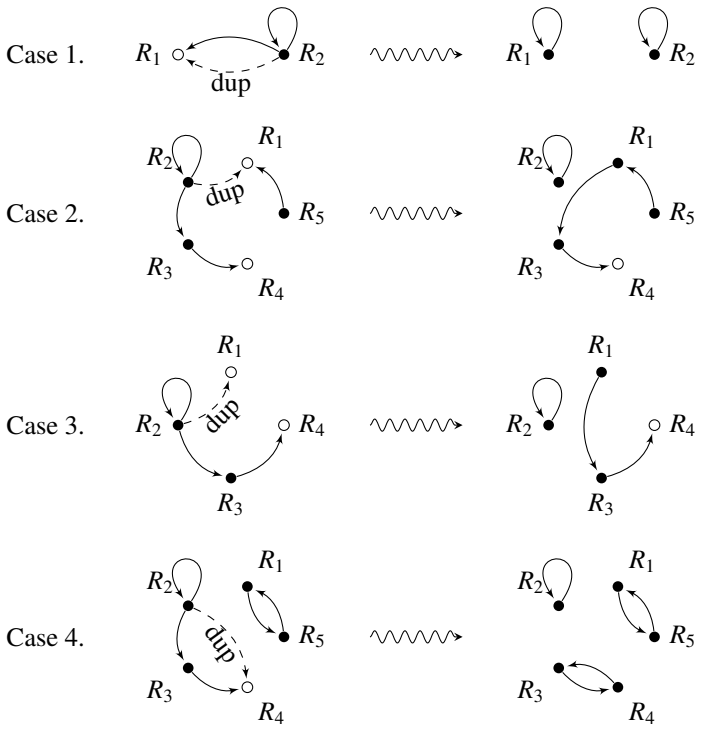
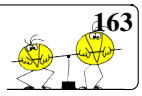


Figure 7.10: The four cases of function `De_compose`. The dashed arrow labeled "dup" shows the choice of *dest* when duplicating *source* register R_2 on the left parallel copies. The graphs on the right show the state of the parallel copy after adding copy "dup" to the predecessor basic block B_s. Empty bullets show registers not holding live variables at the end of B_s.



number of duplications in $/\!/c$. Otherwise, more spilling needs to be done, or the edge on which $/\!/c$ is must be split (we will come back later to this problem in Section 7.4.3). When *dest* is found, a copy to put the value of *source* into *dest* is added at the end of B_s (Line 27). Note that if there are more than one duplication, the order of the copies added to B_s does *not* matter (only registers not alive are defined). Then, $/\!/c$ is modified so that it reflects that, now, the register *dest* contains the value that should be put in *target* by the parallel copy (Line 29).

Function `De_compose` uses Function `is_Reachable`, page 162. This function consists only in doing a traversal from the "*end*" register, following the edges from the register flow graph in the reverse order, to see if the "*start*" register can reach *end* (in the normal order). Since each register has only one entering edge, there is no choice at each step but to traverse the predecessor. The "marked" array is used to avoid infinite loops if reaching a cycle of edges that do not contain the *start* register (in which case *start* obviously cannot reach *end*).

Better algorithms could be devised for the decomposition, using for instance information on the place where $/\!/c$ will be sequentialized. This is not our purpose here. Our purpose is just to show that it *is* possible to decompose parallel copies that contain duplications and put the latter in the predecessor block, not to provide the best way to do it, if there is one.

7.3.2 The problem of moving reversible parallel copies

Now that we got rid of the duplications in our parallel copy, our goal is to move this reversible parallel copy $/\!/c$ out of the edge E from B_s to B_d. We recall that its reverse is denoted $/\!/c^{-1}$. Moving the parallel copy $/\!/c$ down is relatively easy. It can be placed at the top of B_d and $/\!/c^{-1}$ is added to every edge arriving in B_d other than E. This is correct because the sets of variables alive are the same at the beginning of B_d and on every edge arriving in B_d. The fact that $live\_out(/\!/c) = live\_in(/\!/c^{-1})$ and $live\_in(/\!/c) = live\_out(/\!/c^{-1})$ shows that, in the end, the effect is that: if the flow comes from another edge than E, $/\!/c^{-1}$ is followed by $/\!/c$, i.e., the identity is done on the registers alive; and if the flow comes from E, only $/\!/c$ is performed, which is what was expected.

On the contrary, moving $/\!/c$ up is more difficult. We have seen in Figure 7.8 that $/\!/c$ needs to be modified to take the liveness at the end of B_s into account. Indeed, $/\!/c$ was created with the liveness of E in mind, which is a subset of $live\_out(B_s)$. The differences from moving down or up come from the asymmetry of liveness between the source and the destination of edges. As explained, this forces us to be careful in order not to erase any value of a live variable, and not to add useless copies during compensation. While it is possible to augment or project parallel copies when required, we prefer a more elegant solution in which parallel copies are converted to permutations: $/\!/c$ is made a bijection by replacing the every $\perp$ by a well-chosen register in the array representation. This simplifies the process of moving parallel copies and we see how to do it in the next section.

7.3.3 Converting parallel copies to permutations

The previous section leaves us with the problem of how to take liveness into account (when moving up). We propose a solution based on *permutations*. Permutations consider all registers, i.e., each register is the source and the destination of another; when viewing the permutation as a parallel copy, this means that every register is considered

to hold the value of a live variable. Hence, there is no need to worry about erasing live values when moving permutations.

For our implementations, we propose an extended data structure for permutations compared to parallel copies. Permutation will be represented by two R-arrays "`to`" and "`from`." They represent where the value of a register does come *from*, and where the value of a register will go *to*. This was not consistent for parallel copies because of duplications, as in this case, the `to` field cannot contains more than one destination. However, for reversible parallel copies, the same structure could be used in an implementation, in which case the `from` array represent the array $/\!/c$ we directly use in this chapter. In that case:

$$/\!/c.\texttt{to}[i] = j \in \{1,\dots,R\} \cup \bot$$
$$/\!/c.\texttt{from}[i] = j \in \{1,\dots,R\} \cup \bot$$
$$/\!/c.\texttt{from}[i] = j \iff /\!/c.\texttt{to}[j] = i$$

Definition 7.2. A parallel copy is a *permutation* π if it is reversible and verifies:

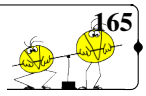
$$\forall i \in \{1,\dots,R\}, \quad \pi.\texttt{to}[i] \neq \bot \quad \text{and} \quad \pi.\texttt{from}[i] \neq \bot$$

We call *de-materialization* the process of creating a permutation from a parallel copy. Since a permutation contains only cycles, chains must be closed to be loops in the permutation. This is possible because we are dealing with reversible parallel copies: every node in the graph representation has *at most* one leaving and one entering edge. Of course there is not a unique solution in the general case. We propose a pseudo-code on Function `De-materialize`.The loop Line 8 finds the register that is at the end of the chain, so that cycles are the smallest. Free registers considered to be in chains of length one, hence self-edges are added for them (which is what is expected). Here, π is created so that there is the maximum number of registers for which π is the identity.

Function `De-materialize(`$/\!/c$`)`

```
   Data: Parallel copy //c.
   Output: Permutation π, a de-materialization of //c.
   /* Make π a copy of //c.                                          */
 1 foreach i ∈ {1,...,R} do π.to[i] ← ⊥;
 2 foreach i ∈ {1,...,R} do
 3 |   π.from[i] ← //c[i];
 4 |   if π.from[i] ≠ ⊥ then π.to[π.from[i]] ← i;
   /* Close the chains by forming the smallest cycles.               */
 5 foreach i ∈ {1,...,R} do
 6 |   if π.from[i] = ⊥ then
 7 |   |   current ← i ;
 8 |   |   while π.to[current] ≠ ⊥ do current ← π.to[current];
 9 |   |   π.from[i] ← current;
10 |   |   π.to[current] ← i;
11 return π;
```

Whenever a parallel copy $/\!/c$ is de-materialized into a permutation π, it is easy to move it up or down. π is simply placed at the bottom of B_s (resp. top of B_d) and



π^{-1} is placed on every other leaving edge of B_s (resp. entering edge of B_d). With our representation, π^{-1} is easily obtained from π since:

$$\begin{aligned} \forall i \in \{1,\dots,R\}, \quad \pi^{-1}.\texttt{to}[i] &= \pi.\texttt{from}[i] \\ \pi^{-1}.\texttt{from}[i] &= \pi.\texttt{to}[i] \end{aligned}$$

The remaining problem will be: how to convert back the permutation π^{-1} into a parallel copy representation? More generally, we will move permutations around and need to be able, for any permutation, to generate the code corresponding to it. There are two problems: First, permutations represent parallel copy instructions while machine instruction are in general sequential (see discussion Section 2.3.6); Second, not all copies are required: registers that are not alive at the point where π is sequentialized should not copy their value into others (although not wrong, it is useless to do it). We solve these problems in the next section.

7.3.4 Sequentializing permutations

Parallel copies are not hardware instructions, and neither are permutations. In the previous section, we managed to find a way to move them out of edges. Now, it remains to *sequentialize* them using actual instructions of the target architecture. This is classically know as the Parallel Assignment Problem in the literature, which is NP-complete if assignments depends on multiple values [see Garey and Johnson, 1979, Problem PO6]. It reduces from Feedback Vertex Set, which amounts to finding the minimum number of edges to remove to break every cycle. In our case, each assignment depends on only one value, hence the graph is very particular (maximum in-degree equals one) and cycles cannot intersect, which makes the problem polynomial [May, 1989]. Sequentializing is of course classical in the out of SSA literature; for instance, Briggs et al. [1998] carefully choose an ordering of `move` instructions to avoid the "swap problem." In some cases, swaps instructions are available or can be emulated,[2] but they are usually more costly than moves so they should be used only if there is no free register.[3] While the algorithm we propose is definitely not new in the literature and quite simple, we include it here for completeness.

The pseudo-code is given by Function `Sequentialize` (page 167) and works as follows. First, "re-materialize" the permutation π into a parallel copy $/\!/c$, by deleting unnecessary edges (Line 4). At the same time is build the set of leaves, i.e., registers that receive a value but whose value is not used (in fact, registers that are neither defined nor used are also considered "leaves"). Then, chains that can be sequentialized (Line 11) by performing the last copies of chains first, i.e., starting from the leaves. A register not alive and not receiving any value is viewed as a chain of size one. The loop Line 10 stops right away for them. Then, only cycles remain. If there was any chain, the variable *free* was defined as the root of a chain. Since the copies of all chains are already generated, the value contained in *free* is indeed not needed anymore and is used to break the cycles at Line 16 into chains, as illustrated in Figure 7.11a.

If there are only cycles (even cycles of size one, i.e., self-edges), there is no free register to break the cycles. We choose to perform swaps to resolve the cycles, using $n-1$ swaps per cycle where n is the size of the cycle (see an example on Figure 7.11b, and the general case on Figure 7.11c). In absence of a swap instruction, and if it cannot

[2] By using three XOR for instance, see details in Chapter 8, Section 8.1.2.1.

[3] Unless on some particular cases, for instance for a cycle of size 2, a swap that costs 2 is better than three moves that cost 1 each.

Function `Sequentialize(`π`,Live)`

Data: Permutation π, array Live of registers alive before the point of sequentialization.
Result: Copies are added to the code.

```
   /* Re-materialize parallel copy from permutation.                    */
 1 leaves ← ∅ ;
 2 foreach i ∈ {1,...,R} do
 3     if ≠ Live[i] then
 4         ∥c[π.to[i]] ← ⊥;
 5         leaves ← leaves ∪ {i};
 6     else
 7         ∥c[π.to[i]] ← i;
   /* Sequentialize remaining parallel copy.                            */
 8 free ← ⊥;
 9 foreach current ∈ leaves do
10     while ∥c[current] ≠ ⊥ do
11         generate copy instruction [R_current ← R_∥c[current]] ;
12     free ← current;
   /* Now, there remains only cycles (if any).                          */
13 foreach current ∈ {1,...,R} do
14     if free ≠ ⊥ then                    /* use freeas a temporary register */
15         pred ← ∥c[current];
16         generate copy instruction [R_free ← R_pred] ;
17         ∥c[current] ← free;
18         current ← pred;
19         while ∥c[current] ≠ ⊥ do
20             generate copy instruction [R_current ← R_∥c[current]] ;
21     else                                                  /* need to perform swaps */
22         pred ← ∥c[current];
23         pred_of_pred ← ∥c[pred];
24         while pred ≠ current do
25             generate swap instruction [(R_pred, R_pred_of_pred) ← (R_pred_of_pred, R_pred)];
26             pred ← pred_of_pred;
27             pred_of_pred ← ∥c[pred];
```

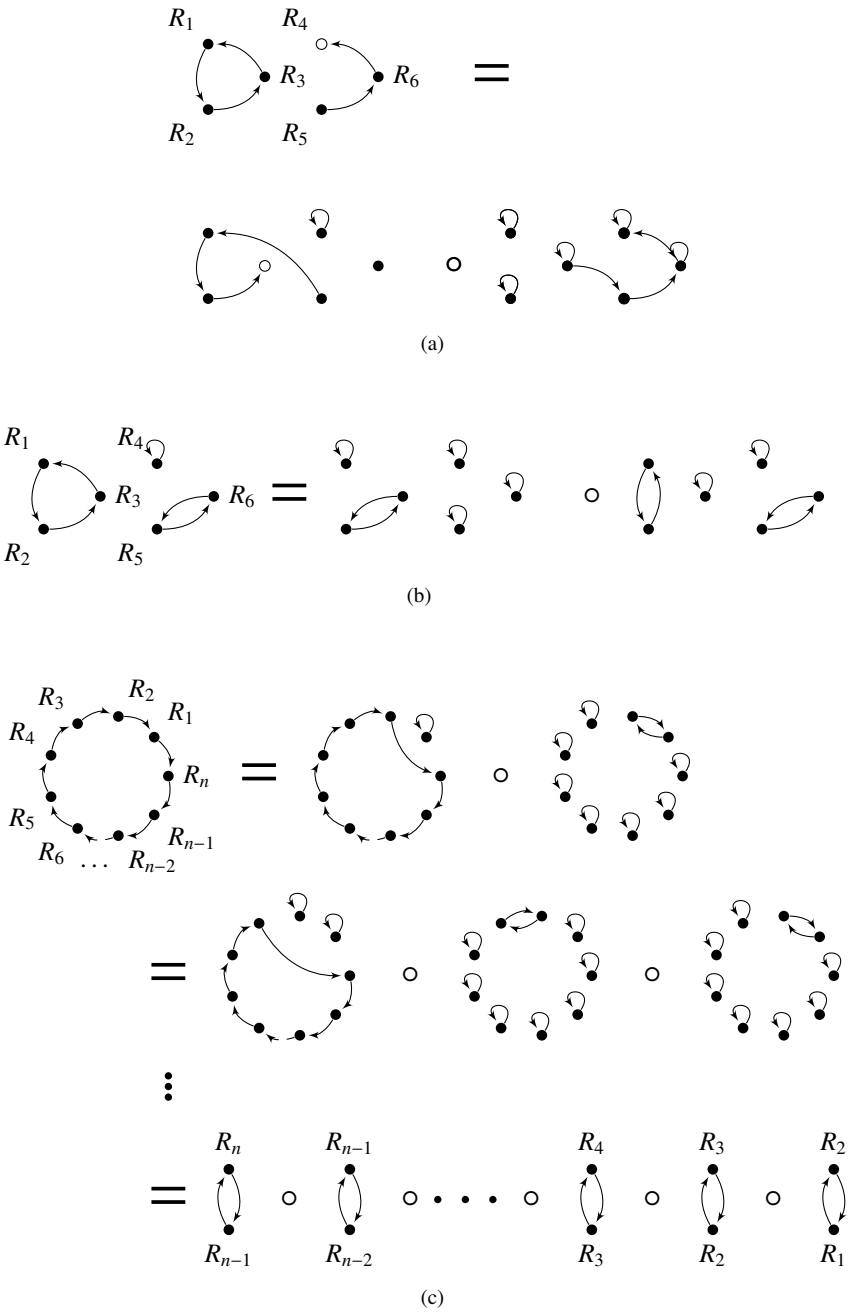


Figure 7.11: Sequentializing reversible parallel copies: (a) after sequentializing the chain $R_5 \to R_6 \to R_4$, R_5 is free and used to break the 3-cycle; (b) there are only cycles and no free register: swaps must be used; (c) general case for a n-cycle when there is no free register.

be emulated, it is possible to spill a register to free it. Then the rest of the cycle can be sequentialized like a chain, and finally the spilled value is reloaded in the right register. However, we will see in the next section that, in practice, parallel copies can be moved deeper inside basic blocks, usually finding program points where there is a free register.

7.4 Put it all together

In the end, the goal of parallel copy motion is to get rid of annoying parallel copies on some edges that one does not want to split. Candidates are critical edges since, for a non-critical edge, it is trivial to move code placed on it—one just moves the code either to the bottom of the source basic block or the top of the destination basic block, depending on which of these has only one entering or leaving edge.

We will not discuss here the problem of choosing which critical edges should not be split. This depends on personal tastes and on the compiler. For instance, profiling can tell that some particular edge is highly executed and should not be split, or some edges have destinations computed on the fly, hence not splittable at compile time. Moreover, in practice, there are edges that we would prefer not to split, but that we would rather split than to pay an insanely large overhead in terms of added moves, or spills. Algorithms should be nicely integrated in one's framework by using cost functions and appropriately deciding whether it is worth to pay extra costs to avoid splitting an edge. We shall not continue this digression here, since our point is the following question:

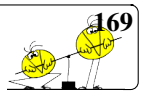
> Given a colored SSA code, a set of edges that cannot be split, and some ϕ-functions, is it possible to remove all ϕ-functions without putting any code on a non-splittable edge and without spilling?

Without loss of generality, we consider that every splittable critical edge has been split, i.e., an empty basic block has been inserted on it. Hence, every remaining critical edge is non-splittable. We have seen in the previous section that knowing if a parallel copy can be moved away from an edge is an easy task. It just depends on the number of duplications in it, which should be less than the number of free registers at the end of the source basic block. The parallel copy can then be decomposed, the duplications being placed on the predecessor block, and the remaining reversible parallel copy is de-materialized into a permutation. The problem is that moving a permutation π up or down requires inserting it reverse π^{-1} on other edges to cancel its effects. Since we now consider several critical edges at a time, sequentializing permutations into compensation code on other critical edges is not a possibility. This means that π^{-1} must also be disposed of.

More generally, the problems that arise when dealing with multiple critical edges and parallel copies at the same time are the following:

- More than one critical edge leaving B_s. Then, there must be at least as many free registers at the bottom of B_s as the total number of duplications summed over all the parallel copies on these critical edges.

- No more free register when sequentializing a parallel copy (obtained from a permutation π) with cycles (at the bottom of B_s or top of B_d). Then swaps can be used, but they are expensive and not always available. We will see that it is possible to move π *inside* the basic block and hopefully find a better place to sequentialize it.



- "Cycles" of critical edges, or long chains of critical edges. Chains of critical edges propagate the compensations, which may be very expensive in the end. Cycles make it impossible to propagate as the compensations would bite their own tails; this is a difficult problem and will be discussed in Section 7.4.2.

The first point falls into the same problem as with only one parallel copy: there must be enough registers, else it is mandatory to split one edge or to spill. The second point was already addressed in the previous section, but we will now see a better way to handle it. Finally, the last point requires full attention and will be discussed last.

7.4.1 Another break in the (permutation) wall

When sequentializing a parallel copy, cycles can exist, like $/\!/c = [R_2, R_3, R_1]$ (see Figure 7.11b). When have seen previously that, usually, when dealing with non-colored variables, a temporary variable is used to store one of the values of the cycle, then all moves can be resolved sequentially. In our case, when variables are already colored, we need a free register for this purpose. If there is none, swaps must be used if one does not want to spill. The choice for this solution depends on how much a load and store will cost compared to the overhead of using swaps instead of just `move` instructions. For instance, a swap costs two moves on a VLIW architecture, hence, if n is the size of the cycle, it costs $2(n-1)$ moves (see previous section, or Figure 7.11c), while the solution with the spill would cost one store, one load, and $n-1$ moves. With a realistic cost model for a VLIW architecture, where a store costs as much as one move, and a load costs 4× more than a move, the solution with the spill is more interesting for cycles of size bigger than six. If swaps are emulated with XOR, swaps costs three instructions and even the smallest cycles are very expensive. Moreover, whenever there are more than one cycle, other cycles benefit from the spilling of one value: this frees a register that can be used as a temporary to break all the cycles.

However, it would be better if there were a free register. So we will not continue the digression, and instead propose another method: the idea is to not only move parallel copies out of edges, but also inside basic blocks. Indeed, when using permutations, moving a permutation π up (for instance) from the bottom of the basic block to a deeper place p corresponds to putting π at p and recoloring the variables from p to the end of the basic block according to π. Figure 7.12 shows an example where it is better to place re-materialize the permutation higher than the bottom of the basic block. Hopefully, we will find a better place to sequentialize π, a place where there is at least one free register.

In fact, this idea can be formalized more cleanly by remarking that, for any region of the program, i.e., any set of instructions, it is possible to add a permutation π at every entry point of the region, to add its inverse π^{-1} at every exit point of the region, and to recolor every variable in the region according to π: if a is assigned to R_i, then assign it to $\pi(R_i)$ on every point of the region. However, there are limitations to this: some instructions have register constraints—e.g., arguments of a `call`—that cannot be recolored. So, unless $\pi(R_i) = R_i$ for all constraints, these instructions cannot be part of such a region.

Using this formalism, it is easy to understand how to move a permutation in a basic block, and more generally how the whole permutation motion works. To move a reversible parallel copy $/\!/c$ out of the edge E from B_s to B_d, let π be a de-materialization of $/\!/c$. If one wants to move $/\!/c$ up (for instance), let us choose any convenient region with an entry point somewhere inside B_s, and exit points on every edge leaving B_s. π is

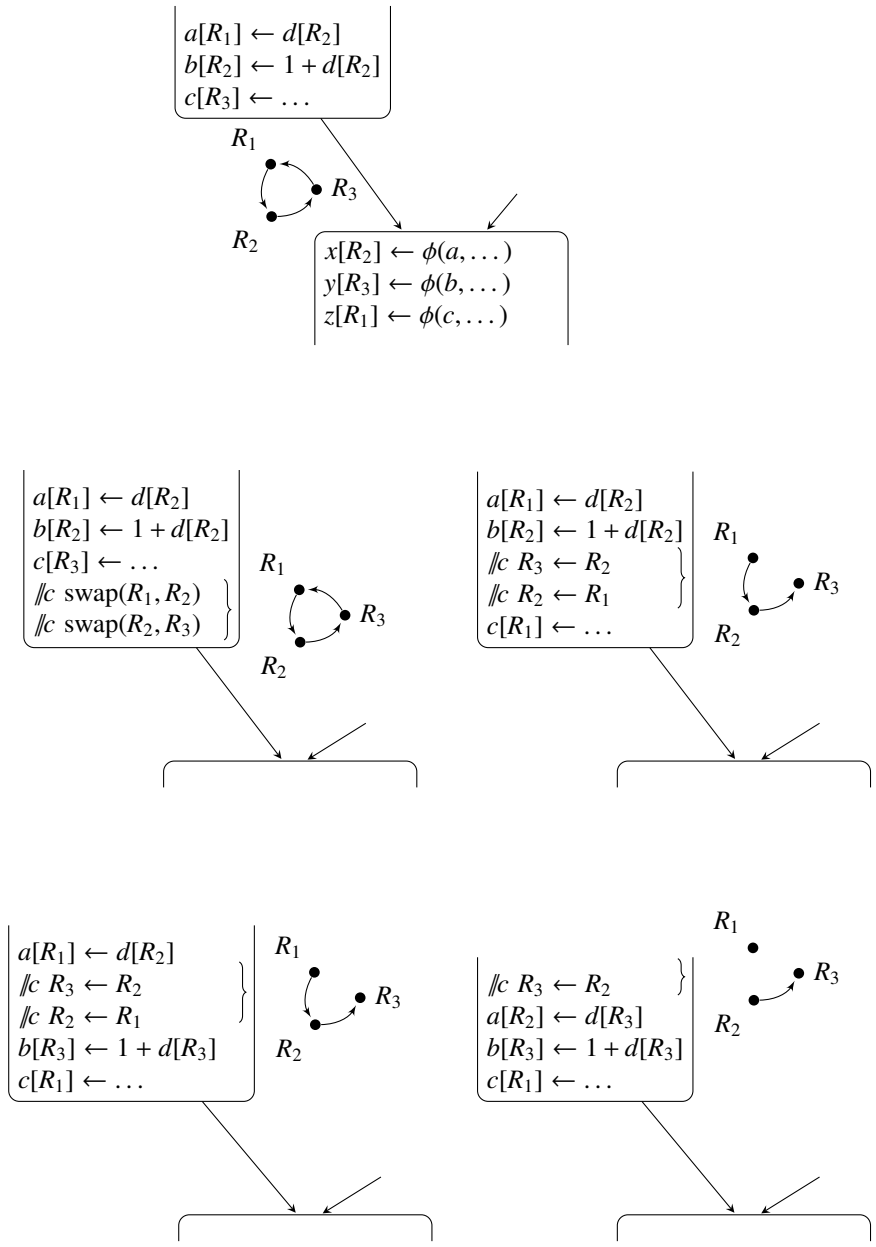


Figure 7.12: Moving a permutation up in a basic block: when sequentialized at the bottom of the basic block, no register is free to break the cycle and it costs two swaps, i.e., 4 moves on a VLIW or 6 XOR if emulated. The permutation can be moved up higher in the basic block, where at least one free register exists. On such point is above the definition of c, where only 2 moves are required for the permutation. Before the definition of a is even better since only 1 move is necessary. The code added by when sequentializing the permutation is preceded by a $/\!/c$ for more visibility.

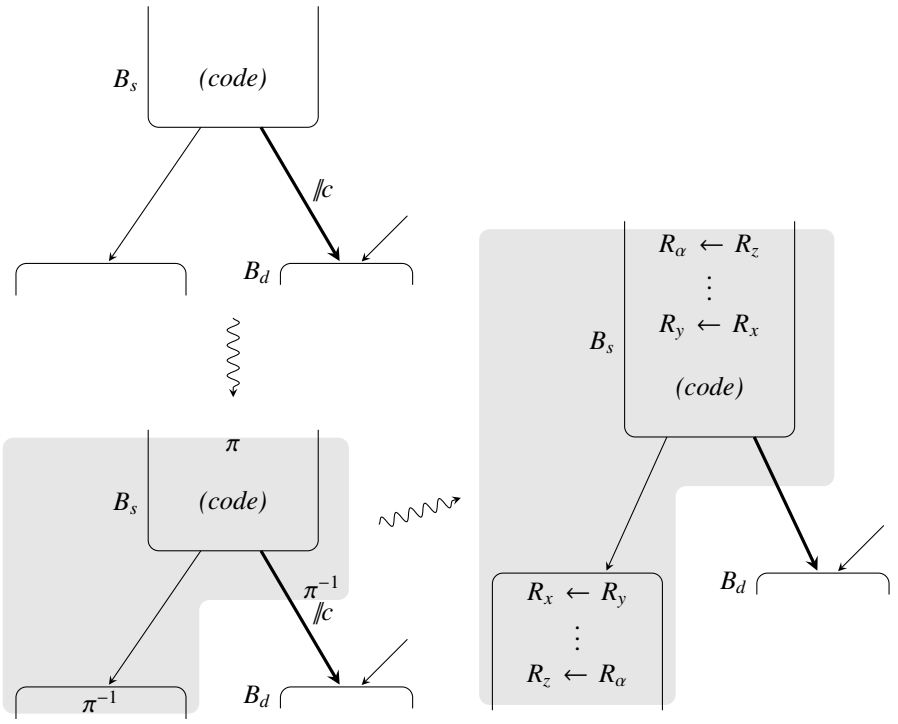


Figure 7.13: Region recoloring: to move up $/\!\!/c$, a region (grayed) is chosen, π is added at the entry, and π^{-1} at the exits. On the critical edge, π^{-1} and $/\!\!/c$ cancel each other. The other entry and exit points can be chosen at convenience inside basic blocks.

added at the entry, and π^{-1} at every exit. On E, π^{-1} and $/\!\!/c$ cancel each other and no code remains. On every other edge, π^{-1} is re-materialized, and π is re-materialized at the convenient point inside B_s. If every other edge is not critical, π^{-1} can be materialized at the top of the corresponding destination basic blocks, or even deeper in the blocks if one finds that some places require less copies. We will discuss the problem that some other edges might be critical in Section 7.4.2.

We call this alternative view of permutation motion *region recoloring*, because it corresponds to recoloring the variables of a region. Shuffle code must be added at the borders to ensure correctness. We think the region recoloring point of view more handy to visualize code transformation. Figure 7.13 shows an example of how to view region recoloring when moving a parallel copy away from a critical edge.

For the choice of the convenient program point inside B_s (resp. B_d), one looks for a point p such that there is no register constraint between p and the end of B_s (resp. beginning of B_d)—or such that π is the identity for these constraints—and where it is cheapest to sequentialize π. This usually means that there should be one free register at p if there are cycles in π, and the number of moves in π that are useless when re-materializing should be maximized—maximize the number of moves in π involving non-live registers.

Good program points to sequentialize the permutation are for instance before or after functions calls: the register pressure is very low just before and after these in-

structions because of crash registers.[4] Hence it is usually not necessary to try to move the permutation beyond calls—along with the fact that very few registers (the callee-saves) are not constrained. If one still wants to go beyond calls, it is always possible to decompose the permutation π into $\pi' \circ \pi_{id}$ such that π_{id} is the identity for all constrained registers. Then, π' is sequentialized while π_{id} can be moved higher in the basic block. We will discuss in Chapter 8, Section 8.2.2.3, situation when this is beneficial.

One useful side-effect of trying to move the permutation upwards (or downwards) is that some remaining moves in the code can be "eaten" along the way, if they are not duplications. So, in order to find the best place, one should start at the bottom of the basic block (if moving upwards) and go up one instruction after the other, without applying any transformation but remembering at each step the cost of placing the permutation there. Whenever a move between registers is found, is it added to the permutation. The process stops whenever a instruction with unmet registers constraints in encountered, or when reaching the other side of the basic block. Then, one knows the best place, and the process can be started again, but now without faking it, to move the permutation to the chosen spot.

> Note: it is possible to view permutation motion as a way to find better split points than the ones SSA provides. Indeed, by comparison, the split everywhere approach of Appel and George [2001] would give low costs to affinities inside basic blocks, and much bigger ones on edges that should not be split. The effect of this would be that the affinities on a particular edge would be coalesced, while those inside basic blocks would remain, i.e., permutations of colors will be placed inside the basic block and on other leaving edges, which is exactly the same effect as permutation motion.
>
> Hence, permutation motion is a way to achieve better results in terms of splitting, close to an aggressive splitting approach but without its drawbacks: there is no enormous interference graph and work can be done directly on the code, linearly.

7.4.2 Chains, trees and butterflies of critical edges

Whenever two critical edges are connected to the entry (or exit) of the same basic block, this poses a problem as, obviously, they cannot move their parallel copy on this basic block since compensation would have to take place on the other edge. Hopefully, it may be possible to move the parallel copies on the other basic blocks attached at the other extremities of these edges. Hence, one should not move parallel copies blindly, locally, without taking other edges into account. One should have a look at other "connected" critical edges, which we define as *siblings*:

Definition 7.3. A *sibling* of an edge E from basic block B_s to basic block B_d is an edge E' so that the source of E' is B_s or the destination of E' is B_d. In the first case, E and E' are called *siblings at top*, in the second case, they are called *siblings at bottom*.

For instance, the two edges leaving B_s on Figure 7.13 are siblings at top. We need a way to decide, for each parallel copy on a critical edge, whether it should be moved up or down. This depends on whether it has siblings or not, which made us define the *weakness* property for critical edges.

Definition 7.4. A *weak* critical edge is a critical edge that do not have any critical sibling at its source or at its destination:

- it is *weak at top* if it does not have any critical sibling at top—i.e., if every other leaving edge of its source basic block is not critical;

[4] The pressure is a bit higher before a `call` than after if arguments are put in crash registers, but is still usually less than R.

- it is *weak at bottom* if it does not have any critical sibling at bottom—i.e., if every other entering edge of its destination basic block is not critical.

See again Figure 7.13; the edge from B_s to B_d is critical, but not the other edge leaving B_s. Hence the critical edge is weak at top. It is now easy to know where to move parallel copies that are on weak edges: if the edge is weak at top, it should be moved up to the source basic block, and if the edge is weak at bottom, it should be moved down to the destination basic block. In both cases, compensation can be added on the siblings edges since they are not critical—the code is in fact added to the basic block at the other end of the edge, which is not a problem since this block has only one entering edge.

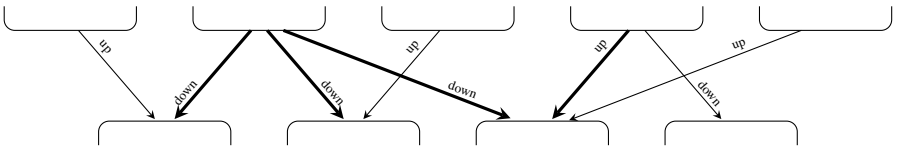
Weakness is transitive and can be propagated: if E is a non-weak critical edge, but has a sibling at top E' that is weak (at bottom), then E is weak at top since it is possible to move its parallel copy on B_s, then to add compensation code on E', i.e., compose the compensation code with the existing parallel copy on E', then move down the resulting parallel copy from E' since it is weak at bottom. Since we are moving permutations, it is an easy task. If $\pi = \pi_2 \circ \pi_1$, then:

$$\begin{aligned} \forall i \in \{1, \ldots, R\}, \quad \pi.\texttt{to}[i] &= \pi_2.\texttt{to}[\pi_1.\texttt{to}[i]] \\ \pi.\texttt{from}[i] &= \pi_1.\texttt{from}[\pi_2.\texttt{from}[i]] \end{aligned}$$

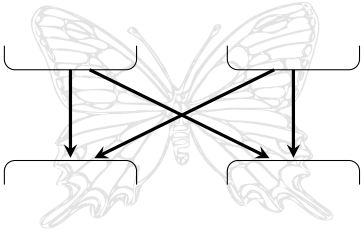
This means that, in absence of cycles between critical edges—i.e., by starting from a critical edge E and following critical edge siblings either at top or bottom, one does not come back to E again—, it is possible to orient all critical edges, for instance by starting from any edge E and then performing a tree traversal from root E. Each critical edge encountered along the way is weak at top if the traversal finds it from a sibling at bottom, and weak at bottom if the traversal finds it from a sibling at top. We suppose that first, all parallel copies where (successfully) decomposed and the duplication placed at the bottom of their predecessor basic block. Then all remaining parallel copies have been de-materialized into permutations. It is not easy to to move all permutations out of the critical edges, by starting from E and "pushing" the permutations along the branches of the tree, re-materializing permutations at every beginning or end of basic block encountered, until the leaves of the tree are reached. Figure 7.14a, shows an example of a tree on which a solution could be to move down all permutations on edges marked as "down," and move up the permutations on the other edges—marked as "up." This motion should of course not be done in any order since compensations might arrive later on edges, but starting from the root of the tree and composing permutations as others are encountered. In our example, we started from the second leftmost source basic block, propagating the constraints from its leaving edges.

> Note: for big trees, one should ask if this is really better that splitting. Indeed, a lot of code is added, at every basic block of the paths of the trees but at the root. In particular, supposing there is only one edge E with a parallel copy (all other copies are the identity), the cost of the whole motion should be compared against the cost of a split (or some spills).

Of course, there is a problem whenever the graph obtained by following critical edge siblings does not form a tree. If the graph has cycles, like the "butterfly" graph of Figure 7.14b, no edge is weak, neither directly nor by transitivity. These edges are called *strong critical edges*, and form bothersome atomic multiplexing regions (see Chapter 3, Section 3.3). If there is a parallel copy (except the identity) on any of the edges of the butterfly of Figure 7.14b, there is no solution involving only motions of parallel copies. We see in the next section that, whenever the permutation motion



(a) Tree



(b) Butterfly

Figure 7.14: Weakness can be propagated along chains of critical edges, and more generally along paths of a tree. (a) all critical edges are weak since there is no cycle; the up/down labels indicate one possibility of motion for parallel copies on edges. (b) critical edges form a cycle, hence none is weak: they are all strong critical edges.

process is stuck, it is always possible to find a solution using standard register allocation techniques.

7.4.3 Whenever permutation motion is stuck

So far, we found two reasons why the permutation motion could fail: whenever there are more duplications than the number of registers not alive at the end of the predecessor block, and whenever control-flow edges form cycles, in which case they are all strong critical edges. The problem is that such situations arise often in real-life programs. Duplications exist whenever two results of ϕ-functions use the same value, and the register pressure can be too high on the predecessor block when this happens. The butterfly of Figure 7.14b is found whenever a loop has a skip edge—for instance, the bottom left and upper right blocks represent respectively the top and bottom part of the same basic block inside the loop, while the top left is the entry and the bottom right is the exit of the loop.

Let us see these problems from the recoloring point of view we introduced in Section 7.4.1. The problem of cycles is the easiest to visualize. See again Figure 7.14b, if there is any permutation on a critical edge of the butterfly, the goal of permutation motion is to *recolor* the entire region defined by the critical edges, i.e., since these edges cannot be split, one tries to insert permutations at the entry and exit points of the region (the bottom of the predecessor blocks and the top of the successor blocks). The region recoloring works if, in the end, the remaining permutations on the critical edges are all equal to the identity. In fact this is exactly the same problem as in the proof of Chaitin et al. [1981]. Indeed, we proved in Chapter 3, Section 3.1.2 that, even with live-range splitting, the proof still holds if critical edges cannot be split. We explained in Section 3.3 that this was linked to *multiplexing regions* , i.e., regions of critical edges that cannot be split, such as our butterfly. The problem with these regions is that they can

define arbitrarily complicated interference graphs. This also creates the problem with the duplications: the two definitions responsible for the duplication interfere, at the bottom of the predecessor block (an entry point of the region), with all other variables alive. If there was R variables alive at the end of the block, this creates an $(R+1)$-clique in the interference graph of the multiplexing region (the argument of the ϕ-functions is not alive in the region but the definitions are, hence there is $R-1+2$ variables at this point).

To solve this problem, we still have one possibility: we can do what people do since Chaitin et al. came up with exactly the same problem, only for much more bigger graphs. The problem is NP-complete, but we can use the same heuristic to perform register allocation on the multiplexing region. Of course, spills might be required, but we do not have much choice. Whenever one gets a solution, stores and loads must be inserted respectively at the entry and exit points of the region if there are spills, and also permutations of colors must be added to repair coloring mismatches. We propose to use the standard Iterated Register Coalescing (IRC) scheme on the interference graph of the region or any Chaitin-like algorithm. The interference graph of the region is easily obtained provided we know which variables are alive and where on a multiplexing region. This was explained in Section 3.3, and we recall in particular that the *definitions* of ϕ-functions are alive in these regions, but not their arguments (unless they are used later, after the ϕ-functions). As for the definition of interference, this time we need to insert the notion of value as in Definition 2.11. Indeed, if a block B has two successors containing the ϕ-functions $[a \leftarrow \phi(c,\ldots)]$ and $[b \leftarrow \phi(c,\ldots)]$, we do not want a and b to interfere at the bottom of B even if they are both alive at this point.

Register allocation of a multiplexing region on an example. An example of register allocation on a butterfly multiplexing region is given on Figure 7.15. Initially, the code is under colored SSA and the ϕ-functions require a swap of R_1 and R_2 on the edge from B_{s_2} to B_{d_2}. We present our example using a different view that is strictly equivalent but might be easier to understand to those familiar with SSA: First, go out of (colored) SSA using the algorithm of Sreedhar et al. [1999], i.e., add copies at the entry and exit points, then rename the copies of variables. In the example, the copies are renamed using primed variables. These variables are not colored and form an interference graph on the butterfly region. Note that in our example, there is no "live-through" variable—variables alive in the region but not involved in a ϕ-function—but these variables *must also be split* at the region boundaries since they must appear in the interference graph. Hence the interference graph of the region is completely disconnected from the rest of the program (from an interference point of view). At the end of B_{s_1}, the couples (A',B'), (A',D'), (B',C') and (C',D') have different values hence interfere, but not the couples (A',C') and (B',D'). However, these last two have different values at the end of B_{s_2} hence interfere nevertheless, and the four variables form a clique. In general, the interference graph can be any graph, as shown by Theorem 3.1. In the interference graph, the register nodes are added as a separate clique so that affinities can guide the register allocation process. These affinities are weighted by the number of moves it would save if the two extremities are assigned to the same color. In our case, none of Briggs's and George's rules for coalescing could coalesce an affinity because of their restriction with registers that are pre-colored nodes (see details in Chapter 6, Section 6.2).[5] In our example, the coloring chooses to spill D' since there is only three registers. Then, it remains to repair the coloring at the region boundaries, i.e., sequentialize the remain-

[5] However, these rules could apply after D' is simplified as a potential spill.

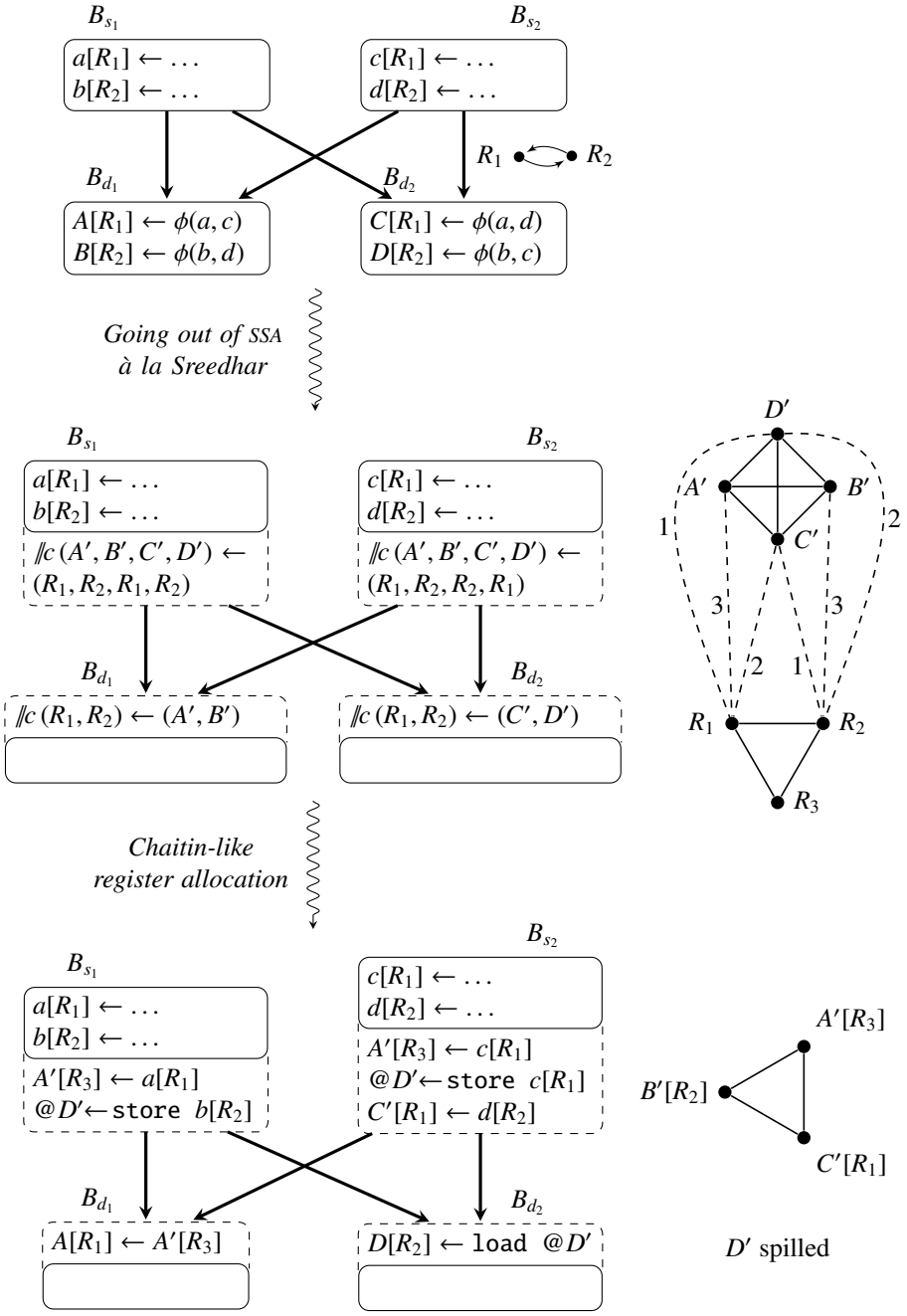


Figure 7.15: Register allocation on multiplexing regions: initially, variables were allocated using only two registers, even if there are three. First, go out-of-SSA *à la* Sreedhar, putting parallel copies at the region boundaries; Second, perform register allocation on the interference graph of the region, trying to coalesce the most affinities. Note that in this example, it would have been better to assign A' to R_1.

ing parallel copies (on the figure, we removed unnecessary copies involving the same register in destination and argument). Note that the copies introduced were parallel, so the order in which they are sequentialized is important; in particular, on B_{s_2}, the copy to A' and store for D' must be executed before erasing the value in R_1 for C'.

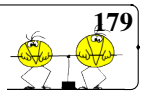
7.5 Conclusion

In this chapter, the goal was to investigate how to go out of a colored SSA code. We have seen that classical techniques to go out-of-SSA that insert copies at the beginning or end of basic blocks, as do Sreedhar et al. [1999], are too constrained in our case by the fact that variables cannot be created on demand. It then depends on whether some registers are free or not. The alternative solution is to place parallel copies corresponding to ϕ-functions on the incoming edges. We wanted to avoid the drawbacks of splitting an edge, and to propose a solution whenever edges cannot be split because of technical problem. Our solution is based on an idea that, to our knowledge, is new in the literature: parallel copies can be *moved* away from edges, provided that *compensation code* is inserted on other edges. Duplications in parallel copies pose some problems when moving them, so we gave an algorithm to decompose parallel copies so that duplications can be handled separately, and the remaining parallel copies do not contain duplications. Our solution is then to convert parallel copies into *permutations* that are easier to move. In fact, we have seen that permutation motion can be viewed, more generally, as *region recoloring*, a technique we introduced that allows permutation to be moved also *inside* basic blocks and not only from control-flow edges. We remarked that this allows us to split at a finer granularity level than the SSA split points, hopefully finding points where it is cheaper to repair coloring than on edges. Finally, we have seen that, in the presence of non-splittable critical edges, the permutation motion can sometimes fail: if the number of duplications exceeds the register pressure and in the presence of *multiplexing regions* as defined in Chapter 3. Since this problem in NP-complete, as shown by Theorem 3.1, we propose to use classical graph coloring techniques of register allocation based on the algorithm of Chaitin [1982] to recolor the multiplexing regions, however possibly with additional spills.

We believe that discovering that parallel copies can be moved is a major breakthrough for out-of-SSA translations. Up to now, it was in general considered that placing copies on edges would require to split them, which is a bad thing to do. For this reason, people tried to introduce copies directly at the basic blocks borders since the discovery of SSA, starting with the algorithm by Cytron et al. [1991]. Recently, the idea of doing register allocation while still under SSA was developed. This allows us to use its nice properties for a longer time, and amongst them the fact that the interference graph is chordal hence easy to color. However, the drawback is that going out-of-SSA introduces parallel copies on edges. Hack and Goos [2008] proposed very recently a recoloring technique used to coalesce the copies on these edges, but still need to split edges whenever the coalescing fails.

To conclude, the work presented in this chapter shows that it is not a problem to place parallel copies on edges as these can be moved away from edges that cannot be split, or that one does not want to split. Moreover, in a context where processors can execute multiple instructions at a time, for instance a VLIW machines, schedulers often have trouble filling completely the issues. This makes a lot of space where to hide additional copies, now that we know how to move them using permutations. This is not possible if copies are added on edges since instruction scheduling cannot spawn

across basic block boundaries. We presented in this chapter a generic method and, in practice, trade-offs must be made between splitting edges or not. Experiments need to be done to confirm the usefulness of the permutation motion method. We will see in the next chapter, conditions which could greatly benefit from this method.



Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth, *Letter to Peter van Emde Boas*

8 Conclusion

In this thesis, we presented many theoretical aspects of register allocation. However, the ultimate goal of compilation optimizations is to be used in practical situations. Reality is different from models: on the one hand, architectures have particular constraints that differs between each other and hence not included in general register allocations schemes; on the other hand, worst-cases situations are seldom found in actual programs, which simplifies some problems in practice. In this conclusion chapter, we revisit the most important points developed in the previous chapters of this thesis and place them in a broader context, to provide insights or advices for register allocation in practice.

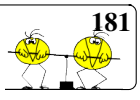
8.1 Reality is different from models

During the few years that this thesis lasted, we worked in collaboration with the compilation team of STMicroelectronics from Grenoble. They provided us with many fruitful discussions, in particular, they brought a valuable expertise of the problems related to the implementation of an industrial strength compiler. Among the architectures they target, the most important one is the ST200, a family of Very Long Instruction Word (VLIW) processors capable of executing up to four instructions at a time (four issues). The compiler they use and develop to compile for this family is called "LAO," for *linear assembly optimizer*. We will now talk about particular architectural constraints that need to be taken into account, referring to, but not limited to, our experience with the ST200.

8.1.1 Architectural constraints complicates register allocation

8.1.1.1 The constraints

There are usually many architectural constraints in a processor, which must all be taken into account. For register allocation, we are interested in two flavours of constraints concerning registers: the register constraints and the naming constraints. The former imposes constraints on the uses of registers, i.e., in which context they can or must be used. The latter imposes constraints when assigning variables to registers. We give now extensive examples based on the ST200, which has 64 integer registers (general register file) and 8 boolean registers (branch register file). It does not however have any floating point register file.



Register constraints. Of the 64 integer registers of the ST200, many are completely equivalent but some are very special. R_0 is a bit bucket, i.e., it represents a constant (zero) and writing in it is possible but has no effect. The thread pointer (TP) is in R_{13} and is not modifiable. The stack pointer (SP) is in R_{12}, can be modified, but must always be valid. Finally, the link register (or return pointer, RP) is in R_{63}. It contains the instruction where execution must continue after a function call returns. It is excluded from some register classes (e.g., cannot hold a memory address to perform a `load`). Other registers have more freedom, but many more conventions exist: the frame pointer (FP) is in R_7, the static link (SL) in R_8, the global pointer (GP) in R_{14}. The application binary interface (ABI) defines constraints so that functions in libraries behave nicely when called: the caller-save registers are: all the branch registers, i.e., B_0 to B_7, and the integer registers R_8 to R_{11} and R_{15} to R_{63}. The callee-save are the others: R_1 to R_7 and R_{14} (since R_0, R_{12} and R_{13} already have a special treatment). Arguments to function calls are passed using registers R_{16} to R_{23}. These serve also when functions returns small structures with less that eight values. Otherwise, R_{15} is used for the return value.

Naming constraints. Here, we talk about general naming constraints found in architectures, not necessarily in the ST200. Such constraints can be: two operands must reside in the same register: case of auto increment or RISC assembly instructions (e.g., $R_x \leftarrow R_x + R_y$). Two operands *must not* reside in the same register (e.g., some architectures forbid to multiply a register by itself). There are also more complicated constraints like register pairing: a 64-bit `load` defines registers R_{2p} and R_{2p+1}; or register aliasing: e.g., two 16-bit registers names point to the "lo" and "hi" part of a 32-bit register. Finally, we also cite constraints on register classes, when the arguments or results of an instruction must reside in a subset of the registers: for instance the integer division on X86 expects the dividend to be on 64 bits on the 32-bit registers `%edx` and `%eax` (the divisor can be any register), and defines these two registers respectively as the remainder and quotient of the operation.

8.1.1.2 Solutions for constraints on registers

We voluntarily gave a long list of constraints, not because it is important to know precisely all of them, but to illustrate the fact that they exist and are not petty details. Still, register allocation algorithms often seem to elude these constraints, which results in elegant strategies where variables of a program can be assigned to any of the R registers available. If register allocation where really performed this way, there would not be many running programs. *And yet they run.* How are these constraints satisfied in practice? Let us start by the register constraints.

Register constraints. Registers that are reserved for a particular purpose are easy to handle. For instance, there cannot be two stack pointers at the same time, so it is possible to rename directly the variable SP by R_{12} in the code. From an interference graph point of view, this corresponds to merging the two nodes. In the case of SP, it is even possible to directly remove R_{12} from the set of assignable registers since it is always alive and it will never be possible to assign any other variable to R_{12} (unless it is a copy of SP on all its live-range). For R_0, this is a bit different since it can have two uses: putting the value zero in a register, or throwing away the result of an instruction. In the first case, it can be directly used in the code. The second case is useful for instance if an instruction defines two variables but only one needs to be kept. If there is only one register left, using R_0 for the useless definition saves a spill. A solution is to

detect definitions that become immediately dead using liveness analysis and to replace the corresponding variables by R_0 in the code. Of course, R_0 should not be in the set of allocatable registers. Finally, the case of R_{63} is particular since it cannot be used to load from memory. This means for instance that in $[\ldots \leftarrow \texttt{load}(a)]$, a cannot be assigned to R_{63}. This can be easily assured by adding an edge between a and R_{63} in the interference graph.

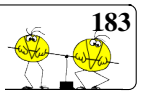
It is tempting to solve ABI constraints like the SP case above. Such a solution would "hard-code" constraints in the program. For instance, supposing variable a is constrained to reside in R_x by the instruction that defines it, then a is replaced by R_x everywhere in the program. Of course this poses a problem if the same instruction is used to define another variable b where a is still alive. In that case different possibilities involving the spilling of a or moving it to another register arise. Obviously, a `move` is better than inserting loads and stores. This is in general the preferred solution, and to save the burden of finding points of conflicts, a safe solution is to insert a copy $[a \leftarrow R_x]$ directly after the defining instruction. Hopefully, the coalescing during register allocation will manage to assign a to R_x and the copy will be removed in the end. If not, this probably means that it was more preferable to save R_x for other variables.

In general, this is the way register constraints are handled: by splitting variables, i.e., adding copies between actual registers and variables, either before or after the instruction. For instance, arguments of functions calls are "put" in their right registers before the `call`, and the register holding the return value copies its content into the right variable after the `call`.

Naming constraints. The above method obviously works also for naming constraints where the architecture imposes the use of one particular register in an instruction, as in the example of the division for x86. However, this method does not work with constraints like register aliasing or pairing. To handle register aliasing, Smith et al. [2004] propose to extend the classical graph coloring technique by generalizing the constraints between nodes in the interference graph, hence giving to each node a set of registers available to it instead of a number of colors. An easy work-around is to consider only disjoint register classes by forcing registers being referenced in more than one class to be part of only one. Note however that this cancels out the benefit of having aliasing in the architecture.

Constraints like pairing cannot be as conveniently left aside. However, they can be easily handled with some splitting. The difference with the previous solutions based on splitting is that splitting only the variables on which the constraint applies is not enough. Indeed, suppose that every odd-numbered register contains a variable, then there is no solution for the `load64`. But if for instance R_2 and R_4 are free, a solution is to save the content of R_3 to R_4, then perform the load into R_2R_3, then swap R_4 and R_3 so that this last register holds the same variable as before. Hence, a solution is to split *all variables* before and after the instruction that needs pairing. This creates a set of variables completely disconnected from the rest of the interference graph, on which it is easy to satisfy the pairing constraint. This solution also relies on the coalescing to minimize the effect of splitting.

Very recently, Pereira and Palsberg [2008] showed how to make register allocation with both register pairing and aliasing work in an easy and straightforward manner for what they call "elementary form," i.e., a program where variables are split at every program point. This further stresses the point of the next section.



8.1.1.3 Splitting even more

Splitting is a very convenient way to deal with architectural constraints on registers. In order to avoid a serious drop of quality of the final code, we need a good coalescing algorithm capable of removing the most number of inserted copies. We studied many variants of this problem in Chapter 5: aggressive, conservative, incremental and de-coalescing, and found most of them NP-complete but for the incremental coalescing on chordal graphs. However, we came up in Chapter 6 with practical heuristics that perform very well, at least for the suite of graphs from the Coalescing Challenge provided by Appel and George [2000]. In particular, we proposed a conservative heuristic based on an improved brute-force test and of existing tests of Briggs and George. This heuristic outperforms the state of the art heuristics that we are aware of, i.e., the conservative Iterated Register Coalescing (IRC) of George and Appel [1996], and the optimistic coalescing of Park and Moon [2004]. However, we do not know how it compares to the very recent recoloring algorithm proposed by Hack and Goos [2008], as explained in the conclusion of Chapter 6. Our results for the graphs of the Coalescing Challenge are important for splitting algorithms, since the graphs have been obtained by aggressive splitting, where live-ranges are split at every program point. This gives an idea of the maximum splitting one could find in a program, and shows that even with that much splitting, our coalescing techniques are good enough to remove most of the copies, leaving only about 10% more than the optimal solution (compared to the ~100% more left by the IRC).

In this thesis, we proved in Chapter 2 that the interference graph of a program under Static Single Assignment (SSA) is chordal (Theorem 2.37). This led to new ideas of performing register allocation while still under SSA to use the fact that chordal graphs are easy to color with the greedy scheme of Chaitin et al. [1981], i.e., are greedy-k-colorable (see Property 2.23). However, architectural constraints, if treated as explained above, break the chordal property of the interference graph. Indeed, this inserts a clique of nodes corresponding to machine registers, which periodically interfere with some variables at every instruction that constrains its operands. This creates cycles in the interference graph that are likely to be of size at least four, and without any chord. This can even make the interference graph not greedy-k-colorable anymore. The solution we advocate is to split the same way as for the pairing problem, i.e., insert a parallel copy *for all variables* alive before the instruction, and also after. Indeed, this creates new variables that are just live-in or live-out of the instruction, or both. These are two intersecting cliques that intersect also with the clique of register nodes. If this is done for every instruction that constrains variables, then the interference graph is made of (at least) two connected components. One of them contains the register nodes and all the variables with tiny live-ranges created, i.e., a union of clique. The remaining part of the graph stays chordal. The component containing the register nodes can however be non-chordal. This is the case for instance if two arguments of an instruction are restricted to different subsets of the same register class. For instance, if a and b are respectively forbidden to reside in R_1 and R_2, then (a, R_1, R_2, b) is a chordless cycle of size four. However, this is not a problem as this component is still easily colorable.

To conclude, most constraints on registers can be solved with the aid of splitting. This allows to keep a simple general register allocation scheme, provided that one is confident about one's coalescing algorithm. This is our case.

8.1.2 Architectural constraints that simplify register allocation

In Chapter 3, we revisited the NP-completeness proof of Chaitin et al. [1981], after the discovery that SSA interference graphs are chordal. The conclusion was that, in fact, the problem of knowing whether R registers was sufficient for register allocation is *easy*, unless critical edges cannot be split or the architecture can define two variables at a time but do not allow swaps. We come back to these problems, taking into account practical considerations.

8.1.2.1 Repairing color mismatches is easy

When the splitting of variables is allowed anywhere, in particular also on critical edges, the intuitive result is that, if there is at most R variables alive at each point of the program, it is possible to color each instruction independently and repair the coloring where there are mismatches. This is of course an inefficient way to do register allocation but the point here is to give the idea of why coloring is not a problem. Problems arise whenever it is hard to repair mismatches. Let us see the different cases. A mismatch occurs whenever a parallel copy is not the identity. As explained in Chapter 7, sequentializing parallel copies is easy if there is no cycles, or if there are cycles but at least one free register. Otherwise, permutations must be used.[1] In the ST200, there is no swap instruction, but the four issues give parallelism that allows to perform permutation cycles of size up to four, hence the swapping is not a problem. Even if the parallelism did not allow to swap, a possibility that works for integer or boolean registers is to use three consecutive XOR instructions. Swapping R_x and R_y is then easily performed by executing:

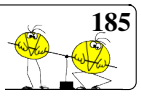
Instructions	R_x	R_y
	a	b
$R_x \leftarrow R_x \oplus R_y$	$a \oplus b$	b
$R_y \leftarrow R_x \oplus R_y$	$a \oplus b$	$b \oplus a \oplus b = a$
$R_x \leftarrow R_x \oplus R_y$	$a \oplus b \oplus a = b$	a

However, note that this technique does not work for any register class, like registers holding floating point numbers for instance.[2] We explained in Chapter 3 that, if swaps cannot be performed, problems can arise if, still, some instruction can define more than one variable at a time. We do not think this is realistic for an architecture, but let us consider that swaps cannot be performed on the ST200. What are the instructions that can define at least two variables? For instance, a `load64`. However, even if it defines two registers, it only takes one as argument, meaning that at least one register was free before the instruction, which allows a swap. The same is true for the function `call`, for which all the caller-save registers are free before and after, but for the argument and result registers (a strict subset, hence at least one free register exists). Hence the only instructions that could potentially pose problem in fact break the difficulty by providing program points where it is easy to repair coloring mismatches.

In conclusion, swapping is not a problem in practice and repairing color mismatches is always possible. There remains however the problem of non-splittable critical edges.

[1] It would also be possible to spill the value of a register to free it, but would mean we did not manage to use only R registers. It would also not prove that this was not possible.

[2] A word of caution in presence of register aliasing: R_x and R_y should be physically different or the value is erased by the first XOR and cannot be recovered. Indeed, $R_x \oplus R_x$ is always equal to zero.



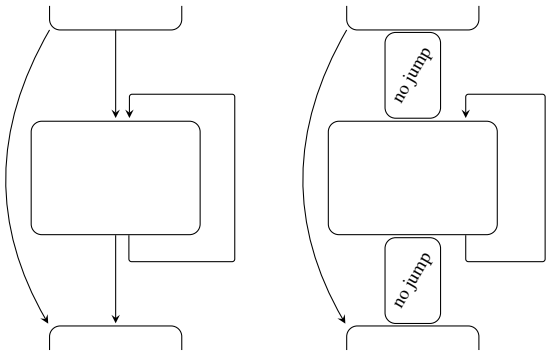


Figure 8.1: In a loop with a skip edge (a), all edges are critical. Since the final code is linearized in memory (b), the edge entering the loop and the one exiting it can be split without adding a `jump`.

8.1.2.2 False critical edges can be split

Both Chapters 3 and 7 state that non-splittable critical edges define atomic multiplexing regions that are hard to color. In Chapter 7, Section 7.4.3, we proposed to use the same heuristic as Chaitin [1982] to perform register allocation on these regions, since the problem is NP-complete. We measured that, in practice, many multiplexing regions are "butterflies," as in the example given on Figure 7.14b. However, note that basic blocks do not exist independently of their machine representation. In this representation, the final code, it is well-know that the control-flow graph (CFG) is linearized: in this linear representation, basic blocks lie one after the other in memory, and we remarked that some critical edges are what we call "false" critical edges.

Definition 8.1. A critical edge from basic block B_s to basic block B_d is *false* if B_s and B_d are consecutive in the linear representation of the CFG.

It is in fact easy to add code on these edges. Indeed, there is no `jump` instruction to go from B_s to B_d: the edge is critical because at the end of B_s a (conditional) jump goes to somewhere else, and the next instruction, the start of B_d, is the destination of a jump. The actual choice of going to B_d from B_s is actually performed by *not choosing to jump*. Hence, if some instructions are added between B_s and B_d in the linear representation, they will automatically be executed whenever the execution path follows the edge. Let us see an example. We already explained that butterflies of critical edges appear whenever a loop has a skip edge. Such a loop is depicted on Figure 8.1a: the four control-flow edges are critical and form a cycle. In the compiler, linearization is likely to place the three basic blocks in memory in the same order as they are represented. In this case, the edge that enters the loop and the edge that exits it are false edges. A basic block can be added to them with no additional jump, as depicted in Figure 8.1b. This breaks the butterfly cycle, and any permutation on the skip edge or the back edge of the loop can be moved up or down, with appropriate compensation code placed on the false critical edges as explained in Chapter 7.

Limitation of false edges. There are however limitations to the usefulness of false edges. First, they usually correspond to highly executed edges in the code: this was

indeed a good reason to put the basic blocks next to each other in the first place: to save one `jump`! So it might not be very sensitive to come later and add compensation code on such an edge and a permutation on one of the two basic blocks, just to save a `jump` on another less executed edge... Second, if some code is added to false edges, it cannot be scheduled with code on other basic blocks as scheduling does not spawn across basic block boundaries. Hence this code cannot be hidden in empty issues of bundles of VLIW architectures. Still, false edges provide a nice trick to get rid of annoying strong critical edges, when one absolutely wants neither to spill nor to split an edge.

We have just seen that practical issues are not always a problem. In the first case, we comprehensively explained why it is not realistic to consider that repairing color mismatches is a problem. The second case remind us that, although it is more elegant to abstract practical problems, one must not forgot the reality to which they are attached to. In this case, it is easy to forgot that control-flow edges are in fact only either a jump instruction, or the *absence* of a jump instruction.

8.2 Register allocation in practice

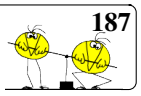
The previous section reminded us that register allocation is for real architectures, with the assortment of constraints that comes shipped with them, which can be bothersome but also sometimes beneficial. We will now give our view of how register allocation should be done.

8.2.1 Global versus local

In this thesis, we advocate that the spilling phase should be separated from the coalescing and coloring phase. We based this thought on the fact that the interference graphs of SSA programs were chordal, hence there is an easy test to know whether spilling is necessary or not. This test is that Maxlive, the maximum number of simultaneously alive variables, should be at most R, the number of registers available. The goal of our complexity study of the spill everywhere under SSA, in Chapter 4, was to discover whether SSA also simplifies this problem, as it does for the coloring problem. However, we came up with nearly only NP-completeness proofs, and no satisfying algorithm to use in practice.

So, to the question whether SSA is good for the spilling, our answer is "no." For years, evidence was shown that SSA simplifies engineering, but we state here that it does not necessarily give the right splitting points for the spilling. SSA does not provide a simple polynomial algorithm, nor does it bring quality to the spill problem.

There is however a morale to this story. Splitting as SSA does simplifies the coloring, but not the spilling. We believe this shows that spilling is not a problem that is easily handled globally, while the coloring and coalescing are. And that is really a point in favor of doing register allocation in two separate phases. The first phase can optimize loads and stores locally, until the Maxlive $\leq R$ test becomes true. Then, live-ranges are split, using SSA split points for instance, and the chordal interference graph can be built so that coalescing and coloring perform the remaining task of register allocation. Spilling is performed as a global approach in the algorithm of Chaitin et al. [1981], or the IRC of George and Appel [1996]. We think the fact it is global is responsible of two weaknesses. First, spilling a node in the interference graph forces the corresponding variable to be spilled "everywhere," while maybe spilling only parts



of it would be sufficient. Second, there is no guarantee that spilling a node will help to color the graph. In the LAO compiler, we ran the following experiment: after the last spilling phase, each spilled variable was submitted to a test. If, on all points of its live-range, the register pressure was at most Maxlive – 1, it was declared as a "useless" spill and inserted back into the code. We found that, even if there was not so many useless spills, there existed some, i.e., too many. Of course, these spills were sometimes useful for the coloring part because there was no SSA-based splitting, but in the proposed context where splitting is done after the spilling phase, this shows that this global approach for spilling is probably not the right one.

8.2.2 Proposed scheme(s)

8.2.2.1 Aggressive scheme

Our experiments from Chapter 6 with the coalescing in presence of aggressive splitting show that the approach of Appel and George [2001] is worth trying. While they stayed with an enormous amount of copies after their optimal spilling formulation, we showed that these can be dealt with using a conservative approach not more complicated than the IRC from an implementation point of view. However, our experiments with the optimal "clique" rule (see Section 6.4) also show that not all split point of a program are needed to find the best locations where colors must be repaired. If the spilling phase does not need a preliminary aggressive splitting phase, it is worth trying to find less points of split, in order to minimize the impact of having to manipulate an very large interference graph. In this direction, it would probably be interesting to compare the split points chosen by our "chordal" conservative coalescing technique (i.e., the affinities not coalesced), to the split points chosen by a technique based on our permutation motion (i.e., the points where permutations are sequentialized).

8.2.2.2 Local spilling followed by coalescing

In this section, we propose two more schemes for register allocations, based on what we have seen so far. The common idea is to use local decisions for spilling. As proposed by Hack et al. [2006], the spilling can be performed using an algorithm similar to the one proposed by Belady [1966], which was initially designed for virtual-storage management. Guo et al. [2004] studied it in the context of local register allocation, i.e., on basic blocks, and showed that it is a good heuristic. Hack et al. describe how to extend it to work in the more general context of CFG, to decrease the register pressure until Maxlive is at most R, the number of registers. They use this algorithm in the context of register allocation under SSA. After the spilling, it is possible to split using SSA split points as they do, or to perform a more aggressive splitting. In our experiments with the ST200, we remarked that, due to the large amount of registers available (64), there is not a lot of move instructions added. Hence, the coalescing does not have a strong impact. In that case, we believe that it is sufficient to split using SSA and use our permutation motion technique described in Chapter 7 to get rid of the remaining copies. In particular, the low register pressure around function calls gives us good hopes that permutations at places where there is no free register can be easily moved in order to avoid performing expensive swaps. As a future work, it would be great to have a good mean of knowing where empty issues are likely to be in the final code so that permutations can be sequentialized at these points. For these reasons, we believe that the overhead of the copies added during the out of colored SSA translation

can be nearly completely nullified.

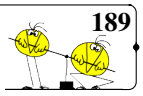
Quentin Colombet worked in our team to incorporate Hack's algorithm in LAO. In order to increase the number of moves to study the effects of our coalescing heuristics, the number of integer registers was artificially decreased to 18 instead of the original 64. To keep the graph chordal, he used the splitting technique we described above, splitting every live variable before and after constraining instructions (mainly function calls). The first experiments showed improvements both in the spilling and the coalescing part, which were compared to an implementation of the IRC. For coalescing experiments, to be fair between the different heuristics, the IRC was applied *after* using the spill algorithm of Hack. We give here some preliminary results for coalescing, based on the "HP Benchsuite." This benchmark suite is provided by Hewlett-Packard and consists of complex but typical C programs. For example, there are programs treating MP2 and MPEG4 streams, DivX conversion, RSA and DES cryptography and the `gcc` source code. We did not run these programs on the ST200, as the experimental branch we are working on is not ready yet. Instead, Quentin Colombet provided us 708 interference graphs of procedures from the HP Benchsuite. These graphs have on average about 250 nodes, and up to 4000 for the biggest graph. Remember that, compared to the graphs from the Coalescing Challenge (which have about 850 nodes on average), these graphs are obtained after splitting at SSA split points and not at any program points. So, the graphs from the HP Benchsuite are of a decent size.

The table below shows how different coalescing schemes perform compared to the IRC. To obtain these numbers, we divided the cost of the heuristic solution by the cost of the IRC solution. The result was then averaged over all graphs, weighted by the size of the graphs so that bigger graphs gets more importance. As many graphs are quite small and easy to coalesce, we also gave the ratio on the set of "interesting graphs," i.e., for each heuristic, the graphs for which the solution of IRC and the heuristic differ. Note that we could not include results for the heuristic of Appel and George based on Park and Moon [2004] since our graphs are greedy-k-colorable, while their heuristic works only for graphs such that every node has degree at most $k-1$. It still works for some graphs, but fails on too many of them to give a fair comparison (however, the few results we have for it are not encouraging, compared to our heuristics).

	Chordal	Clique rule+ chordal	Aggressive+ de-coalescing+chordal
on all graphs	0.8080	0.8066	1.0247
on interesting graphs	0.4748	0.4724	1.0601
#interesting graphs	141	142	188

We see that our chordal rule manages to remove one fifth of the weights of the affinities not coalesced by IRC. And on graphs where improvement was possible, it removed more that half of those left by IRC. As expected, the optimal "clique" rule is not of much help here, since the graphs are not aggressively split. A more surprising result is that the aggressive scheme followed by de-coalescing and chordal performed, in average, *worse* than IRC. A quick investigation showed that some graphs are up to 18× worse that IRC, i.e., the aggressive part made really unfavorable choices that the de-coalescing part could not recover.

To conclude this section, preliminary results seem to indicate that decomposing register allocation in two phases improves the resulting code. Moreover, we already

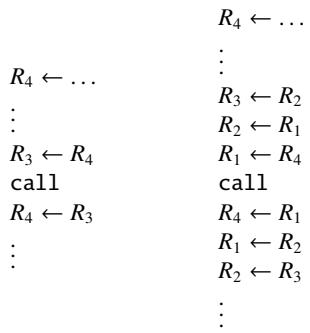


experienced that it really simplifies the development of improvements as each phase is cleanly separated from the other. This allows us to fairly compare different algorithm, for instance by choosing either Hack's algorithm or IRC for the spilling, and then choosing either IRC, any of our coalescing algorithms from Chapter 6, or our permutation motion from Chapter 7. It would have been much more difficult to plug our different coalescing algorithms directly inside the IRC.

8.2.2.3 A few more words on permutation motion

During this thesis, we started to implement permutation motion in LAO. Quentin Colombet continued this work and, although it is not finished yet, we already noticed some interesting facts. We devised permutation motion in order to have a local approach that would move parallel copies inconveniently placed on critical edges. Compared to a global approach involving aggressive splitting and coalescing, it has the advantages of being able to work directly on the code, without the need for big data structures like an interference graph. Practical application include in particular just-in-time (JIT) compilation, where linear algorithms are preferred over more time-consuming approaches. Following this idea, the goal is to apply permutation motion after a fast coloring algorithm, hence local, that has the drawback of missing many coalescings.

To emulate this "bad" coloring, we used the simplification scheme of Chaitin et al., but followed only by a biased coloring. Since our graphs are chordal (we are still under SSA), the simplification in fact corresponds to an on-line coloring that would follow the dominance tree, i.e., a "tree-scan" algorithm, similar to a linear scan algorithm, but working on a tree instead of a linearized program. We remarked that such a coloring was very poor around instructions where we needed to split before and after, like functions calls for instance. Let us see an example adapted from a real situation. We suppose four registers, and that register R_4 holds an argument for a function call. This register is "crashed" by the function, i.e., is caller-save, and we suppose we still need its value later. Hence it should be saved into a callee-save register. We suppose that the three other registers are callee-save, and that both R_1 and R_2 contain variables but not R_3. The right choice to do would be to save R_4 into R_3 and restore it after the call, hence performing the parallel copy $[R_1, R_2, R_4, \perp]$ before the call, and its reverse, $[R_1, R_2, \perp, R_3]$, after the call, as shown below on the code on the left. But, instead, the biased coloring forces the parallel copy to be $/\!/c = [R_4, R_1, R_2, \perp]$, i.e., the code on the right.



This is obviously a problem, as there is three times too much `move` instructions added. Currently, as explained in Chapter 7, the motion of a permutation π stops at instructions where there are constraints for which π is not the identity. This example motivates the need for permutations to be decomposed so that they can traverse function calls and repair such obvious mistakes. Let us see how it would work on this example. Suppose

a permutation, initially the identity, is being moved up. It "eats" the moves along its way up (provided they are not duplications, as most copies in our example) as follows:

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$R_1 \leftarrow R_4$
`call`
$R_4 \leftarrow R_1$
$R_1 \leftarrow R_2$
$R_2 \leftarrow R_3$
$\pi = [R_1, R_2, R_3, R_4]$
$\vdots$

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$R_1 \leftarrow R_4$
`call`
$R_4 \leftarrow R_1$
$R_1 \leftarrow R_2$
$\pi = [R_1, R_3, R_2, R_4]$
$\vdots$

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$R_1 \leftarrow R_4$
`call`
$R_4 \leftarrow R_1$
$\pi = [R_2, R_3, R_1, R_4]$
$\vdots$

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$R_1 \leftarrow R_4$
`call`
$\pi = [R_2, R_3, R_4, R_1]$
$\vdots$

Then, π is stuck because R_4, on which the register constraint is, is not the identity. Indeed, it is not to be possible to "recolor" the argument of the call from R_4 to R_1 since the ABI does not allow it. The solution is to decompose π into $\pi' \circ \pi_{id}$ where π_{id} is the identity for all constraints of the call, as in the code below. Then, π_{id} can traverse the `call`, and the copies above can be incorporated in it. Note that, since R_4 is saved but is also used as argument of the call, the instruction $[R_1 \leftarrow R_4]$ before the call is a duplication and the value in R_4 should not be erased. Hence, π_{id} must not incorporate this instruction but instead *recolor* it.

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$R_1 \leftarrow R_4$
$\pi_{id} = [R_2, R_3, R_1, R_4]$
`call`
$\pi' = [R_1, R_2, R_4, R_3]$
$\vdots$

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$R_2 \leftarrow R_1$
$\pi_{id} = [R_2, R_3, R_1, R_4]$
$R_3 \leftarrow R_4$
`call`
$\pi' = [R_1, R_2, R_4, R_3]$
$\vdots$

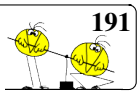
$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_2$
$\pi_{id} = [R_1, R_3, R_2, R_4]$
$R_3 \leftarrow R_4$
`call`
$\pi' = [R_1, R_2, R_4, R_3]$
$\vdots$

$R_4 \leftarrow \ldots$
$\vdots$
$\pi_{id} = [R_1, R_2, R_3, R_4]$
$R_3 \leftarrow R_4$
`call`
$\pi' = [R_1, R_2, R_4, R_3]$
$\vdots$

In the end, the permutation at the top is the identity for all registers, while the one stuck below the call is a swap of R_3 and R_4. Since, only R_3 is alive after the call, sequentializing these permutations gives:

$R_4 \leftarrow \ldots$
$\vdots$
$R_3 \leftarrow R_4$
`call`
$R_4 \leftarrow R_3$
$\vdots$

This example shows that permutation motion should not be restricted to parallel copies on control-flow edges, but also for any parallel copy created for instance to help solving register constraints. Converted to permutations, it is easy to implement



the composition of permutations as explained in Chapter 7, which allows to remove unnecessary copies as the ones in the above example.

8.2.2.4 Towards JIT compilation

Permutation motion is one contribution of this thesis that can be really helpful in a JIT context. It allows to do coalescing, and can work locally in linear time while the coalescing solutions we proposed in Chapter 6 run in quadratic time. We think that, in general, separating register allocation in two phases is an important step for JIT. This allows to have an independent spill that does not have to take care of later coloring or coalescing. Approaches where part of the compilation is made off-line are currently being explored in the JIT world. This allows for instance to use aggressive, time-consuming strategies off-line. Then, their results for spilling can be encoded in the generic code to serve as an oracle for the on-line register allocation algorithm. In this context, an off-line spill that relies on a good coloring technique is problematic, as the on-line coloring will not manage to be as good, hence will require more spill. It is costly to add annotations to the code, and having also an oracle for the on-line coloring is not the best way to solve this problem. But if we know the interference graph is chordal at the time the off-line compilation is performed, we know that any greedy algorithm *à la* Chaitin will manage to color it. We indeed proved, in Chapter 2, the Property 2.23, which states k-chordal graphs are a sub-class of greedy-k-colorable graphs. Hence, even the on-line coloring algorithm will be capable of coloring it, which makes the spill oracle much more useful.

Finally, for JIT context where there might potentially be really many moves, we do not know yet if the permutation motion will be efficient enough. In such a situation, an aggressive coalescing approach allows to remove copies very quickly in a first time. Then, it might be possible to reuse our de-coalescing algorithm, which can be applied to greedy-k-colorable graphs. In the end, if there is not enough time to perform a good de-coalescing, it would probably still be interesting to save many copies, possibly at the price of a few more spilling. All this is hypothetical at the time of this writing, but we believe there is interesting work to do in this direction.

8.3 Conclusion

Since their publication, the graph coloring algorithm of Chaitin et al. [1981], and then the register allocation scheme of Chaitin [1982] (that includes spilling) have been very popular. They are not however the only existing graph coloring algorithms, and serious propositions of other algorithms have been made, for instance by Chow and Hennessy [1990], Callahan and Koblenz [1991] or Lueh et al. [2000]. However, our impression is that the schemes that are actually used are mainly those that flow directly from the algorithm of Chaitin, for instance the Chaitin-Briggs allocator of Briggs et al. [1994], or the Iterated Register Coalescing of George and Appel [1996]. We believe that this is because these allocators manage to keep things simple, which is probably at least as important as producing good register allocated code. Other allocators are conceptually pleasant, but apparently scare too much the developers since very few actual implementation are reported in the literature, and one had to wait for instance more than ten or twenty years before Cooper et al. [2005] and Cooper et al. [2008] compared two of the previously cited algorithm to a Chaitin-Briggs one.

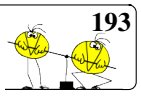
While Chaitin-like algorithms are simple from a theoretical point of view, the classic way of performing spilling, splitting, coalescing and coloring in only one phase is a drawback when it comes to improving one or more of these components. Because the spilling depends on the coloring and *vice versa*, improvements on register allocation tend to be very intricate. While is it usually possible to add one's improvement or another to an existing Chaitin-like framework, trying to mix two or more of them rapidly becomes a nightmare. Then, it seems that "interesting but complicated" improvements are likely to share the same fate as alternative graph coloring algorithms: they get appreciated by the community but are actually seldom used in compilers. A "good" reason for that is also that, while in theory register allocation can be pretty clean and nice, real architectures makes the actual implementation less simpler, as explained at the beginning of this chapter. *A fortiori*, a register algorithm that is already complicated in theory has good chances to be nightmarish to implement.

We believe that this is one of the reasons we see people turning to linear scan algorithms. This extreme way of modeling the register allocation problem suppresses many problems, and allows people to find "optimal" allocations. These are of course optimal only in this modeling, but improving the linear scan framework, which is still quite new, is easy both from a theoretical and an implementation point of view. The morale is to keep things simple, as even if an algorithm performs better than another, the latter is likely to be preferred if it is simple to understand and to implement and if, on the contrary, the former requires years of careful engineering. In these conditions, important improvements like live-range splitting as proposed by Cooper and Simpson [1998] have a hard time to make their way into compilers.

Hence, we believe that one of the challenges for register allocation is to transform the base scheme laid by Chaitin [1982] into an scheme as simple but that also allows improvements to be simple. This is the main reason that we advocate register allocation to move to schemes separated in two phases. In these schemes, the easy test of whether Maxlive is greater than the number of registers R or not controls the spilling. When this is assured, one knows that some splitting will always be enough to color the interference graph later. To illustrate the benefit of using a scheme in two phases, we will take the live-range splitting example. Fabri [1979] already remarked what splitting allows to spill less. Briggs [1992] included a form of aggressive live-range splitting in his compiler, but this created too many copies in the programs. Finally, Cooper and Simpson [1998] decided to split live-ranges on demand, whenever the decision to make a spill is going to be made. This produced really better results, but complicated the Chaitin-Briggs scheme a little more, as the technique must be inserted with spilling on the one side and coalescing on the other side. These two problems have opposite interests when in comes to live-range splitting.

What about live-range splitting in the context of register allocation in two phases? It is true that splitting variables can avoid some spilling, however, it is not of much help whenever there are more variables alive than the number of available registers: in that case, spilling is mandatory.[3] Since the goal of the first phase is to make sure that Maxlive gets lower than R, any splitting might be used if it helps finding a better load-store optimization. For the second phase, we know that splitting as SSA does is enough for the coloring to get feasible. So, if the first phase did its job, there should not be any problem at the second. Now, what happens if the first phase added blindly a lot of copies as did Briggs in this thesis? We have already seen that the most aggressive way of splitting is as Appel and George [2001] do, and that our chordal conservative

[3] Note that re-materialization can also help in fact.

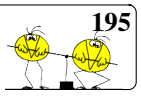


coalescing technique introduced in Chapter 6 can cope satisfactorily with the copies introduced. This shows that, if spilling is always considered worse than adding copies, it is safe to optimize first the spilling without fear of introducing too many copies, then to optimize the coalescing using good conservative techniques. By separating register allocation in two phases, the problem of making a trade-off between splitting too much and splitting too less is not relevant anymore, so the whole scheme gets simpler.

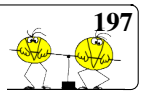
The believe the work done during this thesis to be an important step in favor of the decomposition of register allocation into two separated phases: first spilling, then splitting and coloring using coalescing. Chapter 2 presented important foundations for this thesis: the proof that interference graphs of programs under SSA are chordal, and the proof that chordal graph are greedy-colorable, i.e., colorable using the greedy scheme of Chaitin et al. [1981]. In Chapter 3, we felt that the NP-completeness proof of Chaitin et al. needed to be revisited to clarify where the complexity of register allocation does come from. We proved that knowing whether R register are sufficient to allocate a program or not is not difficult because of the coloring, but because of the presence of non-splittable critical edges. This explained why SSA simplifies the coloring problem: it implicitly supposes all edges to be splittable. This study established solid grounds that showed register allocation in two phases was possible.

Then, we wondered whether SSA also simplifies the spilling problem, and studied the complexity of the simpler spill "everywhere" problem under SSA in Chapter 4. However, we found most versions of this problem to be NP-complete, which makes us think that spilling is not a problem that should be handled globally, but rather locally, to better optimize loads and stores. In register allocation in two phases, the coalescing problem is now a fully independent problem. Its complexity was never studied in details before and we think that the thorough study of the variants of this problem in Chapter 5 was a necessary step. Unfortunately, most of these problems were also proven NP-complete, but for the particular case of incremental coalescing on a chordal graph. However, keeping a graph k-chordal after an incremental coalescing constrains the graph too much, and we preferred to use this result as a heuristic. We used it for the incremental problem on greedy-k-colorable graphs in Chapter 6, since this problem is still open. However, even if the algorithmic behind this heuristic is nice, it appeared to be of little improvement compared to our coalescing heuristic based on "brute-force." We indeed showed in this same chapter that, instead of testing affinities several times as in the IRC, it was more beneficial to perform a more exact test, but only once. The quality of the final coalescing is greatly improved, however at the price of a slowdown of the algorithm. We also improved optimistic coalescing techniques so that they can be applied to any greedy-k-colorable graph, but the results are still not as good as our best conservative technique. This showed that, contrary to common thought, optimistic coalescing is not always the best solution compared to conservative coalescing. We also gave evidence using integer linear programming (ILP) that this last strategy, based on incremental, will be hard to improve and that the limiting factor is the order in which affinities are considered. More work should be done in this area to investigate which ordering works best and why. We believe our work on coalescing heuristics to be of major importance as it canceled the last reason to perform register allocation in only one phase, which was that coalescing techniques were not good enough to cope with many copies. Our results were obtained using the graph database from the Coalescing Challenge [Appel and George, 2000], but we gave previously in this chapter preliminary results based on programs from the HP Benchsuite that confirms the first results.

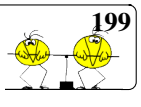
Finally, we investigated the problem of parallel copies being placed on control-flow edges during the translation out of colored SSA. This situation arises during the second phase of register allocation, when the coloring is performed under SSA, for instance to use the chordal property of the interference graph. We developed in Chapter 7 a method based on permutation motion to move parallel copies out of the edges that cannot be split. We also remarked that this method can be viewed as *region recoloring*. We think this method has good chances of being very useful as a post-pass in a JIT compiler, to repair coloring mistakes in a linear and local fashion. For VLIW architectures, permutation motion is also likely to be appreciated if studies manages to predict where empty issues can be found, so the permutation can be moved at these places and copies will not use more machine cycles.



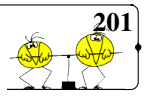
List of Figures



List of Tables



List of Algorithms



Bibliography

F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19 (3):137, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360018.360025.

B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-252-7.

A. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-58274-1.

A. Appel and L. George. Optimal Coalescing Challenge. http://www.cs.princeton.edu/~appel/coalesce, 2000.

A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of the acm sigplan conference on programming language design and implementation*, pages 243–253. ACM Press, 2001. ISBN 1-58113-414-2.

R. Barik, C. Grothoff, R. Gupta, V. Pandit, and R. Udupa. Optimal bitwise register allocation using integer linear programming. In *Languages and Compilers for Parallel Computing*, volume 4382/2007 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-72520-6. doi: 10.1007/978-3-540-72521-3_20.

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

C. Berge. *Graphs and Hypergraphs*. North Holland, 1973.

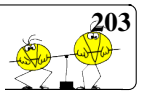
P. Bergner, P. Dahl, D. Engebretsen, and M. T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989. ISBN 0-89791-306-X.

S. Blazy and B. Robillard. Live-Range Unsplitting for Faster Optimal Coalescing. Technical report, Centre d'Étude et de Recherche en Informatique du Cnam (CEDRIC), 2008.

B. Boissinot, S. Hack, D. Grund, B. D. de Dinechin, and F. Rastello. Fast liveness checking for SSA-form programs. In *CGO'08: proceedings of the sixth annual ieee/acm international symposium on code generation and optimization*, pages 35–44, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org/10.1145/1356058.1356064.

B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-ssa translation for correctness, efficiency, and speed. In *International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society Press, March 2009.



F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, Aug. 2005.

F. Bouchez, A. Darte, and F. Rastello. Advanced conservative and optimistic register coalescing. In *CASES'08: Proceedings of the 2008 international conference on Compilers, +Architectures and Synthesis for Embedded Systems*, pages 147–156, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-469-0. doi: http://doi.acm.org/10.1145/1450095.1450119.

D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for riscs. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 122–131, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: http://doi.acm.org/10.1145/106972.106986.

P. Briggs. *Register allocation via graph coloring*. Phd thesis, Rice university, Apr. 1992.

P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the conference on Programming language design and implementation*, pages 275–284. ACM Press, 1989.

P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, 1998.

P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.

Z. Budimlić, K. Cooper, T. Harvey, K. Kennedy, T. Oberg, and S. Reeves. Fast copy coalescing and live range identification. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'02)*, pages 25–32, Berlin, Germany, June 2002. ACM Press.

D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113462.

G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction (CC'82)*, volume 17(6) of *SIGPLAN Notices*, pages 98–105, 1982.

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

F. Chow and J. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Not.*, 19(6):222–232, 1984. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/502949.502896.

F. Chow and J. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, Oct. 1990.

K. Cooper, A. Dasgupta, and J. Eckhardt. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, Oct. 2005.

K. D. Cooper and A. Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 39–49. IEEE Computer Society, 2006.

K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 174–187. Springer Verlag, 1998.

K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/504709.504710.

K. D. Cooper, T. J. Harvey, and D. M. Peixotto. Chow and Hennessy vs. Chaitin-Briggs Register Allocation: Using Adaptive Compilation to Fairly Compare Algorithms. In *SMART*, 2008.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

R. Cytron and J. 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, Aug. 1987.

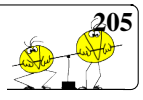
R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. In *PLDI'93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 36–45, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-598-4.

R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *24th Annual ACM STOC*, pages 241–251, Victoria, Canada, 1992. ACM Press.

David W. Goodwin and Kent D. Wilken. Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming. *Software: Practice and Experience*, 26 (8):929–965, 1996.

C. Eisenbeis, F. Gasperoni, and U. Schwiegelshohn. Allocating registers in multiple-instruction issuing processors. Rapport de Recherche 2628, INRIA, 1995.



J. Fabri. Automatic storage optimization. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 83–91, 1979. ISBN 0-89791-002-8.

M. Farach-Colton and V. Liberatore. On local register allocation. *Journal of Algorithms*, 37(1):37–65, 2000.

J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

S. M. Freudenberger, T. R. Gross, and P. G. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Trans. Program. Lang. Syst.*, 16(4):1156–1214, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/183432.183446.

C. Fu and K. Wilken. A faster optimal register allocator. *Microarchitecture, IEEE/ACM International Symposium on*, 0:245, 2002. ISSN 1072-4451. doi: http://doi.ieeecomputersociety.org/10.1109/MICRO.2002.1176254.

D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math*, 15(3):835–855, 1965.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal of Algebraic Discrete Methods*, 1 (2):216–227, 1980.

L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 11–16, New York, NY, USA, 1986. ACM. ISBN 0-89791-197-0. doi: http://doi.acm.org/10.1145/12276.13312.

M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM. ISBN 0-89791-272-1. doi: http://doi.acm.org/10.1145/55364.55407.

D. W. Goodwin and K. D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(199608)26:8<929::AID-SPE40>3.3.CO;2-K.

D. Grund and S. Hack. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In *Compiler Construction*, pages 111–125, July 2007. doi: 10.1007/978-3-540-71229-9_8.

J. Guo, M. J. Garzarán, and D. Padua. The power of belady's algorithm in register allocation for long basic blocks. In *Languages and Compilers for Parallel Computing*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 374–390. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21199-0. doi: 10.1007/b95707.

S. Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, Oct. 2007.

S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

S. Hack and G. Goos. Copy coalescing by graph recoloring. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 227–237, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375610.

S. Hack, D. Grund, and G. Goos. Towards register allocation for programs in SSA-form. Technical Report RR2005-27, Universität Karlsruhe, Sept. 2005.

S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*. Springer Verlag, 2006.

M. Hailperin. Comparing conservative coalescing criteria. *ACM Trans. Program. Lang. Syst.*, 27(3):571–582, 2005. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1065887.1065894.

L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321312.321317.

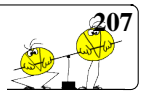
S. Kannan and T. Proebsting. Register allocation in structured programs. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–368, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics. ISBN 0-89871-349-8.

A. B. Kempe. On the Geographical Problem of the Four Colours. *American Journal of Mathematics*, 2(3):193–200, Sept. 1879.

D.-H. Kim and H.-J. Lee. Integrated instruction scheduling and fine-grain register allocation for embedded processors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4017 of *Lecture Notes in Computer Science*, pages 269–278. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-36410-8. doi: 10.1007/11796435_28.

S. Kim and S.-M. Moon. Rotating register allocation for enhanced pipeline scheduling. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 60–72, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: http://dx.doi.org/10.1109/PACT.2007.61.

K. Knobe and K. Zadeck. Register allocation using control trees. Technical Report No. CS-92-13, Brown University, 1992.



Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software: Practice and Experience*, 33(11):1003–1034, 2003.

T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for java 6. *ACM Transactions on Architure and Code Optimization*, 5(1):1–32, 2008. doi: http://doi.acm.org/10.1145/1369396.1370017.

A. Leung and L. George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 204–214. ACM Press, 1999a.

A. Leung and L. George. A new MLRISC register allocator. Technical report, New York University, 1999b.

V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *8th International Conference on Compiler Construction (CC'99), held as part of ETAPS'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 137–152, Amsterdam, The Netherlands, Mar. 1999. Springer Verlag.

G.-Y. Lueh, T. Gross, and A.-R. Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22(3):431–470, 2000.

C. May. The parallel assignment problem redefined. *IEEE Transactions on Software Engineering*, 15(6):821–824, 1989. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.24735.

S. G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. In *Compiler Construction*, volume 4420/2007 of *Lecture Notes in Computer Science*, pages 126–140. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-71228-2. doi: 10.1007/978-3-540-71229-9_9.

M. Naik and J. Palsberg. Compiling with code-size constraints. *Trans. on Embedded Computing Sys.*, 3(1):163–181, 2004. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/972627.972635.

Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: http://doi.acm.org/10.1145/158511.158519.

C. Norris and L. L. Pollock. Register allocation over the program dependence graph. *SIGPLAN Not.*, 29(6):266–277, 1994. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/773473.178427.

J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.

J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, pages 196–204. IEEE Press, 1998.

F. M. Pereira and J. Palsberg. Ssa elimination after register allocation. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 158–173, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7. doi: http://dx.doi.org/10.1007/978-3-642-00722-4_12.

F. M. Q. Pereira and J. Palsberg. Register allocation after classical SSA elimination is NP-complete. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS'06)*, Vienna, Austria, Mar. 2006.

F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 315–329, Tsukuba, Japan, Nov. 2005.

F. M. Q. Pereira and J. Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375609.

M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: a system for fast, flexible, and high-level dynamic code generation. *SIGPLAN Not.*, 32(5):109–121, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258926.

F. Rastello, F. de Ferrière, and C. Guillon. Optimizing translation out of SSA using renaming constraints. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–278. IEEE Computer Society, 2004.

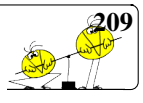
F. Rastello, F. de Ferrière, and C. Guillon. Optimizing the translation out-of-SSA with renaming constraints. Technical Report RR2005-34, LIP, ENS Lyon, France, august 2005.

L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.

H. Rong, A. Douillet, and G. R. Gao. Register allocation for software pipelined multi-dimensional loops. *SIGPLAN Not.*, 40(6):154–167, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065030.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73562.

V. Sarkar and R. Barik. Extended linear scan: An alternate foundation for global register allocation. In *Compiler Construction*, volume 4420/2007 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-71228-2. doi: 10.1007/978-3-540-71229-9_10.



M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996875.

V. C. Sreedhar, R. D. Ju, D. M. Gillies, and V. Santhanam. Translating out of Static Single Assignment form. In A. Cortesi and G. Filé, editors, *Proceedings of the 6th international Symposium on Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 194–210. Springer Verlag, 1999.

S. Touati and C. Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2), June 2004. World Scientific.

O. Traub, G. H. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 142–151, 1998.

S. R. Vegdahl. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'99)*, pages 150–154, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-094-5.

M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. ISSN 0164-0925.

C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the first International Conference on Virtual Execution Environments (VEE)*, 2005.

M. 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.

M. Yannakakis and F. Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987. ISSN 0020-0190.

A. P. Yershov. Alpha—an automatic programming system of high efficiency. *J. ACM*, 13(1):17–24, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321312.321314.

Index

Page numbers are underlined in the index when they represent the definition or the main source of information about whatever is being indexed. A page number is given in *italics* when that page contains an instructive example, use, or discussion of the concept in question. It is said that "A good, professionally written index is a road map that leads both experts and novices in your field to every pertinent sentence you have written." This is not a good, professionally written index.