Algorithmique du décalage d'instructions

Guillaume Huard

N° d'ordre : 201

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le titre de

Docteur de l'école normale supérieure de Lyon
spécialité : informatique
au titre de l'école doctorale de mathématiques
et informatique fondamentale

par **Guillaume HUARD**

Algorithmique du décalage d'instructions

Présentée et soutenue publiquement le 6 décembre 2001

Après avis de :
Madame Christine EISENBEIS
Monsieur Jean-Claude KÖNIG

Devant la commission formée de :
Monsieur Patrice QUINTON (président du jury)
Madame Christine EISENBEIS
Monsieur Jean-Claude KÖNIG
Monsieur Robert SCHREIBER
Monsieur Alain DARTE (directeur de thèse)

Thèse préparée à l'école normale supérieure de Lyon
au sein du laboratoire de l'informatique du parallélisme

Remerciements

Tout d'abord merci à Valentine qui a été tour à tour source de soutien, de préoccupation, de joie et beaucoup plus encore. Sans elle ce travail ne serait sûrement pas le même. Merci de m'avoir accompagné durant ces trois années de dur labeur et de m'avoir aidé à en arriver là.

Merci à Alain qui, à force de persécution, a fini par me faire accoucher de la plus belle thèse du monde ;-). Malgré sa disponibilité faillible et ses excursions sporadiques hors du territoire, il a su, grâce à sa persévérance, nous pousser tous les deux jusqu'au bout de chaque nouveau défi scientifique. Merci à son éternel souci de perfection sans lequel je n'aurais pas pu être aussi satisfait de ce document.

Merci à Christine et Jean-Claude qui ont eu la patience de relire ma prose, et merci à Alain, Christine, Jean-Claude, Rob et Patrice qui ont bien voulu juger mon travail. Merci pour leurs remarques pertinentes et leur enthousiasme.

Merci à Antoine, toujours disponible pour tous, toujours là en cas de besoin. Il a été durant ces trois années un ami avec qui j'ai non seulement festoyé mais également travaillé et enseigné.

Merci à Anne pour la confiance qu'elle a eue en moi aussi bien dans un cadre de recherche que dans un cadre d'enseignement. Merci à Fabien pour sa grande capacité d'écoute et la pertinence de ses questions. Merci à Georges et Frédéric pour avoir tracé la voie avant moi. Merci à Yves pour son éternel pragmatisme, sa pertinence et sa capacité à déstabiliser Alain. Merci à Anne-Pascale et Sylvie pour m'avoir supporté dans mon incompétence administrative de chercheur. Merci à Antoine, Fabien, Vincent et Frédéric pour m'avoir supporté dans leur bureau. Merci à Gabriel pour sa grande curiosité et son intérêt pour mon travail. Merci à Sid pour sa constante bonne humeur et pour nos nombreuses discussions. Merci à Loïc, avec qui j'aurai pu travailleri, pour son aide technique. Merci à Antoine, Anne, Fabrice, Pascal et Stéphane avec qui j'ai souvent passé de longues heures de préparation pour mes enseignements. Merci à Jean-Claude sans qui je n'aurais jamais mis les pieds à Lyon. Merci à Bélaïd avec qui j'ai commencé à apprendre le métier de chercheur. Merci au LIP qui pendant trois ans a été une seconde maison. Merci aux membres du CAR Group pour leur dynamisme et leur amabilité. Grâce à eux, j'ai fait le lien entre vie réelle et recherche théorique. Merci à N. et E. pour avoir su être une source constante d'amusement et un éternel exemple à ne pas suivre.

Enfin merci à ma famille, à mes amis et à tous ceux qui n'ont pas participé directement à ce travail. Merci aussi à tous ceux que j'ai (immanquablement) oubliés.

Avant propos

Dans cette thèse, j'ai tâché autant que possible d'éviter les anglicismes communs à notre discipline. Bien souvent pourtant, lorsque la traduction n'est pas évidente ou littérale, je rappelle le terme anglais en même temps que la première apparition de sa traduction. Plus grave encore, j'ai conservé certains termes en anglais ! De mémoire je n'en vois que deux : *retiming* et *maxlive*. Le premier est un terme avec lequel j'ai vécu pendant trois ans, pour lequel aucune traduction ne m'a paru satisfaisante. Bien sûr, au fil de cette thèse, le lecteur pourra glaner l'expression « décalage d'instructions » qui est une bonne manière d'exprimer le concept de *retiming* dans le contexte de la compilation. Cependant le *retiming* est plus général, et des traductions comme « resynchronisation » ou « retemporisation » sont à la fois inesthétiques et imprécises. Finalement, c'est en trouvant l'anglicisme *timing* dans un dictionnaire français que j'ai choisi de conserver le terme original. Pour le cas de *maxlive*, j'avoue être plus coupable. La seule raison pour laquelle j'ai conservé le terme anglais est qu'il désigne une quantité précise et que, dans ce contexte, une traduction me choquait car elle me donnait l'impression de parler d'autre chose. J'espère que le lecteur saura me pardonner ces quelques entorses à la langue française.

Table des matières

Introduction

Aujourd'hui, l'ordinateur est devenu un produit de consommation à grande échelle ; comme la télévision il a maintenant sa place dans la plupart des foyers et est largement utilisé de manière professionnelle dans tous les secteurs d'activité. Le plus surprenant est son évolution phénoménale depuis son apparition que l'on mesure généralement par l'évolution de la fréquence d'horloge de son processeur. De façon schématique, nous pourrions comparer cette fréquence d'horloge au rythme cardiaque du corps humain : à chaque « top » d'horloge le processeur est capable de réaliser une opération élémentaire. Intuitivement, une idée largement répandue est que plus un processeur est cadencé à une fréquence d'horloge élevée, plus il exécute rapidement un calcul. Pourtant la réalité va à l'encontre de cette intuition : un processeur donné peut très bien être plus rapide qu'un autre processeur cadencé à une fréquence d'horloge plus élevée. Ce phénomène étrange nous révèle simplement que la performance d'une machine ne se résume pas à la fréquence d'horloge de son processeur. Effectivement, plus la fréquence d'horloge d'un processeur est haute, plus il sera à même d'effectuer une grande quantité de travail « élémentaire » en un temps donné. Pourtant, rien ne garantit qu'à chaque « top » d'horloge un tel travail soit disponible, ni que deux processeurs ayant une architecture différente effectuent un travail « élémentaire » équivalent.

Mais si cette fréquence d'horloge n'est pas le critère de choix, quels paramètres faut-il considérer pour juger de l'efficacité d'une machine ? Comme nous pouvons nous en douter, cette question est complexe et tous les éléments d'une machine ont un rôle à jouer dans son efficacité. En fait, même la notion d'efficacité d'une machine est relativement floue, et il est clair depuis longtemps que la façon dont est programmée une application est déterminante sur sa vitesse d'exécution sur une architecture donnée. La question se déclinerait donc sous deux formes finalement équivalentes : pourquoi deux machines différentes ne se comportent pas de la même manière lors de l'exécution d'une application ? Et pourquoi deux réalisations différentes d'une application ne se comportent pas de la même manière lors de l'exécution sur une architecture donnée ? Afin de trouver des éléments de réponse à cette question épineuse, considérons un exemple concret qui illustre à merveille notre propos : la récente sortie du Pentium 4 d'Intel.

Rarement un processeur n'a fait l'objet d'autant de controverses que la première version du Pentium 4 d'Intel. Mis en comparaison dès sa sortie avec d'une part son prédécesseur le Pentium III et d'autre part son concurrent l'Athlon d'AMD, il fait tout d'abord pâle figure lors des premières mesures de performances et se voit opposer un grand nombre de détracteurs de sa nouvelle architecture ([85, 84, 79]). Que s'est-il donc passé ? Pourquoi un nouveau processeur exécutant le même jeu d'instructions que ses deux éléments de comparaison à des fréquences d'horloge beaucoup plus élevées met-il plus de temps à terminer un même calcul ? La réponse se trouve dans deux éléments fondamentaux concernant le Pentium 4. Premièrement, son architecture marque un tournant radical par rapport à ses prédécesseurs[1], en effet le Pentium 4 a été spécifiquement conçu pour atteindre des fréquences d'horloge plus hautes (voir [60]) ceci grâce à des circuits logiques plus simples, avec

[1] Les Pentiums Pro, II et III qui tous basés sur la même architecture connue sous le nom de P6.

moins d'unités parallèles et à un pipeline beaucoup plus profond qui atteint 20 étages (voir [101] pour une comparaison avec le G4 de Motorola). Deuxièmement, le Pentium 4 dispose d'une bande passante mémoire énorme dont on ne tire pleinement profit que lorsque l'on utilise ses nouvelles instructions (particulièrement les extensions SSE2, voir [51, 52]). Le problème est que l'évaluation de performances d'un nouveau processeur se fait généralement grâce à des programmes compilés de façon générique pour les processeurs d'une ou plusieurs générations précédentes[2]. Autrement dit, ces programmes ne sont conçus ni dans le but d'éviter les faiblesses architecturales d'un nouveau processeur ni dans le but de tirer avantage de ses nouvelles capacités. Pour le Pentium 4, il n'est pas difficile d'imaginer la difficulté qu'il peut y avoir à utiliser un pipeline de 20 étages (pénalité de branchement). De plus, dans ce processeur, certaines opérations (comme le décalage logique), jusqu'alors très efficaces dans les modèles précédents, sont désormais plus lentes. Ce qui laisse penser qu'un programme réécrit ou recompilé spécifiquement pour ce processeur pourrait donner des résultats complètement différents de ceux produits par sa version générique. En effet, dans [86] nous pouvons constater qu'un programme d'encodage MPEG4 réécrit par les ingénieurs d'Intel spécifiquement pour le Pentium 4 lui permet de dépasser en performance tous ses concurrents. De façon analogue, la même opération exécutée par les ingénieurs d'AMD afin d'optimiser l'encodage pour l'Athlon permet là encore de gagner en performances (voir [87]).

Cet exemple nous montre que la façon dont est écrit un logiciel a une influence majeure sur son exécution sur une architecture donnée. L'exemple du Pentium 4 n'est pas une exception, tous les processeurs sont sensibles à la manière dont le logiciel est écrit, et ce phénomène ne fait que s'accroître au fur et à mesure que les processeurs deviennent plus complexes (superscalaires, pipelinés, avec renommage de registres, prédiction, spéculation, mémoire cache sur le processeur lui-même, ...). De manière générale, nous pensons donc qu'il est fondamental, pour toute application demandant des hautes performances, d'adapter le logiciel au matériel sur lequel il doit être exécuté. Une solution naïve serait donc de réécrire le logiciel pour chaque architecture pour laquelle il est destiné. Évidemment cette solution est inimaginable en termes de coût de développement et, de toutes manières, est trop complexe pour être mise en œuvre sur de gros logiciels. L'alternative est de disposer d'un compilateur spécifiquement optimisé pour chaque architecture afin de pouvoir, par recompilation du code source, tirer le meilleur parti de chaque machine disponible[3]. Un tel compilateur optimiseur devra idéalement utiliser les instructions les plus appropriées dans le jeu du processeur cible, mais également transformer le programme source afin de l'adapter à la manière dont sont traitées les instructions (comme indiqué dans de nombreux guides d'optimisation, voir [6, 58, 60, 59]). Bien qu'un optimiseur de langage machine puisse réaliser la première partie en remplaçant certaines instructions par un équivalent plus efficace, les problèmes de transformation de code sont intrinsèquement liés aux compilateurs. En effet ils tirent généralement parti de la structure de plus haut niveau du langage source afin d'influer sur la traduction du code en langage machine.

Dans le monde de la recherche, le domaine de la compilation existe depuis bien longtemps, et l'importance du compilateur dans la performance du résultat est bien connue. Ceci est d'autant plus vrai pour les machines parallèles, pour lesquelles l'organisation des calculs et le placement des données en mémoire sont des éléments cruciaux jouant sur la vitesse d'exécution. Si aujourd'hui le rôle du compilateur est également devenu primordial pour les processeurs du grand public, c'est en grande partie dû aux capacités de parallélisme dont ils disposent (plusieurs unités fonction-

[2]Ce qui est logique, puisque la plupart des programmes qu'un processeur aura à exécuter sont compilés ainsi, à commencer par le système d'exploitation : Windows et la plupart des distributions de Linux sont compilés pour le jeu d'instructions du 80386.

[3]Si nous revenons à l'exemple du Pentium 4, [50] traite de l'optimisation du calcul flottant pour ce processeur (par vectorisation) dans les compilateurs actuels.

nelles concurrentes, plusieurs étages de pipeline). Aujourd'hui, ces processeurs peuvent à juste titre être considérés comme des machines parallèles, et les problèmes de compilation qu'ils rencontrent ne sont rien d'autre que les problèmes de compilation rencontrés dans le domaine de la compilation pour machines parallèles. Depuis leur apparition, les machines parallèles ont toujours été accompagnées de compilateurs spécifiques permettant d'une part d'exprimer le parallélisme dans un langage de haut niveau et d'autre part de transformer automatiquement le programme source, afin, par exemple, de faire apparaître du parallélisme, d'améliorer les accès mémoire, etc. (voir les livres [66, 12, 109, 110, 25]). Tout comme la compilation pour les machines du grand public, la compilation pour machines parallèles doit remplir deux objectifs : produire un code utilisant les instructions les plus efficaces dans le jeu mis à la disposition du processeur, et appliquer des transformations au programme source afin d'améliorer son comportement vis-à-vis des différents composants matériels de la machine. Alors que le premier objectif est une optimisation que nous pouvons qualifier de bas niveau, spécifique à chaque processeur, le second revient souvent à utiliser des techniques de transformation générales parfois paramétrées pour produire un résultat adapté à l'architecture. Dans cette thèse, nous ne souhaitons pas réaliser un travail spécifique à un processeur ou à une architecture donnée, nous ne nous intéresserons pas aux optimisations de bas niveau. En revanche nous souhaitons étudier certaines transformations de programme, réalisées durant la chaîne de compilation, afin de déterminer quels avantages elles peuvent apporter lors de l'exécution indépendamment de l'architecture considérée (entre autres nous ne souhaitons pas étudier spécifiquement le Pentium 4, bien qu'il illustre bien le problème de la compilation).

Notre travail s'est porté sur une technique particulière, le décalage d'instructions, qui est un élément de base des transformations de boucles. Cette transformation est souvent utilisée sous diverses dénominations aussi bien pour la parallélisation (seule ou combinée à d'autres techniques) que pour l'alignement de données ou l'optimisation mémoire. Pourtant, le décalage d'instructions est souvent utilisé de manière naïve sans réelle connaissance de toutes les possibilités qu'il peut offrir. Notre but n'est pas de réinventer le décalage d'instructions, mais plutôt de comprendre son rôle dans les transformations existantes, de tâcher d'appréhender ses possibilités, ses limites et son pouvoir d'expression, d'unifier son utilisation sous un même formalisme et de cerner les difficultés liées à son utilisation. Notons que dans ce cadre nous ne cherchons donc pas de réponse optimale aux problèmes classiques de compilation, mais nous cherchons à identifier les problèmes difficiles tout comme les possibilités inexploitées dans l'utilisation du décalage d'instructions pour la transformation de programme. En termes plus spécifiques à l'informatique, cette identification se traduit par l'étude algorithmique des problèmes de décalage d'instructions dans le contexte de la compilation. Ceci donnera lieu, par exemple, à une étude de la complexité des problèmes rencontrés, trop souvent inconnue, et selon le cas à des résultats de NP-complétude (comme celui du chapitre 4 qui va à l'encontre de l'algorithme polynomial présenté dans [69]), ou des méthodes de résolution polynomiales. Pour les problèmes les plus durs, notre but est aussi de donner des moyens constructifs de résolution et éventuellement des heuristiques, afin de mieux comprendre la mécanique mise en œuvre par le décalage d'instructions.

Le point de départ inévitable pour parler du décalage d'instructions dans le cadre de la transformation de programmes est le pipeline logiciel et ce sera l'objet du chapitre 2. Il s'agit d'une transformation de boucle visant à faire apparaître du parallélisme entre les instructions élémentaires qui la constituent (correspondant à des instructions en langage machine). Cette transformation déjà largement connue (voir [67, 93, 2] pour les fondateurs) a non seulement été conçue pour les processeurs spécialisés dans le calcul scientifique (processeurs vectoriels ou VLIW), mais est également adaptée aux processeurs du grand public (voir [6, 58, 60]). De plus le pipeline logiciel est plus que jamais d'actualité, il est le concept de base derrière l'architecture du futur Itanium d'Intel, processeur qui

propose des mécanismes matériels pour le supporter (voir [59]). Le principe du pipeline logiciel est d'exécuter simultanément plusieurs itérations de la boucle en commençant une nouvelle itération avant d'avoir fini l'itération courante, d'où le nom de pipeline logiciel[4]. Le plus surprenant dans le pipeline logiciel décrit sous sa forme classique est qu'il fait un usage intensif mais dissimulé du décalage d'instructions (voir chapitre 2). Nous revenons donc dans un premier temps sur les concepts théoriques masqués derrière le pipeline logiciel et son lien profond avec le décalage d'instructions et la compaction de boucle. Nous passons ensuite à une reformulation dans un modèle plus général du pipeline logiciel décomposé tel que présenté dans [10] et nous proposons un nouvel algorithme de graphe pour la réalisation d'une heuristique proposée dans ce même article. Enfin, nous menons un large nombre d'études statistiques sur l'influence du décalage d'instructions sur la compaction de boucle et la qualité de l'approche décomposée.

L'utilisation d'une technique de pipeline logiciel afin de faire apparaître du parallélisme au niveau des instructions conduit souvent à augmenter dans le même temps le nombre de registres utilisés par une boucle. Non pas que le pipeline logiciel rende l'affectation de registres plus difficile pour le compilateur, mais le parallélisme créé ainsi que l'ordonnancement choisi, conduisent souvent à un plus grand nombre de valeurs intermédiaires à stocker. L'idée première est bien évidemment d'adapter notre algorithme de pipeline logiciel afin qu'il minimise la consommation de registres en même temps qu'il maximise le parallélisme. Cependant, le problème du pipeline logiciel étant déjà NP-complet, ajouter une optimisation supplémentaire le rend d'autant plus difficile à résoudre, en pratique les solutions optimales par programmation linéaire dépassent très rapidement la capacité de nombreux solveurs. Une approche alternative, plus simple que l'approche générale (car elle a moins de contraintes à satisfaire), est de retoucher légèrement l'ordonnancement produit par pipeline logiciel en lui appliquant un décalage bien choisi : c'est le réordonnancement par étages (*stage scheduling*). Une mise en œuvre à base d'heuristiques ou de résolution exponentielle de cette technique est déjà connue (voir [33, 36]). Cependant, la complexité du problème reste toujours inconnue et il est donc difficile de juger de la pertinence de l'approche existante. Nous montrons au chapitre 3 que le problème est NP-complet au sens fort. Ce résultat négatif peut conduire à s'interroger sur l'intérêt du réordonnancement par étages : pourquoi ne pas résoudre directement le problème général de pipeline logiciel minimisant la consommation de registres, lui aussi NP-complet ? La réponse est simple : dans le réordonnancement par étages les ressources sont déjà allouées, ce qui réduit considérablement le nombre de contraintes à satisfaire. La conséquence directe est que la méthode de résolution à base de programmation linéaire que nous proposons (et qui améliore celle de [36]) se révèle très efficace en pratique. De plus, afin de traiter des problèmes de très grande taille, nous proposons également une nouvelle méthode polynomiale d'approximation du problème général.

Un autre domaine dans lequel le décalage d'instructions apparaît est celui de la parallélisation de nids de boucles qui fait l'objet du chapitre 4. De façon classique, il intervient comme constante additionnelle dans l'ordonnancement linéaire décalé et ses variantes plus générales (voir [24, 27, 41, 42]). Ici encore, notre but n'est pas de proposer une nouvelle méthode de parallélisation de boucles, mais plutôt de savoir dans quelle mesure il est possible de contrôler le résultat fourni par les algorithmes connus. En effet, généralement, les méthodes de parallélisation existantes ont peu de contrôle sur la solution produite. Celle-ci est trouvée par programmation linéaire, ce qui rend difficile l'optimisation simultanée d'autres objectifs comme la localité ou l'alignement des données. Contrôler la partie décalage permettrait d'une part de rendre la détermination de la solution moins imprévisible, d'autre part d'atteindre des solutions généralement ignorées par les algorithmes classiques[5]. Le problème

[4]Par référence au pipeline matériel qui consiste à commencer l'exécution d'une nouvelle instruction avant la fin de l'exécution de l'instruction courante.

[5]En effet, la plupart des algorithmes de parallélisation cherchent à « porter » toutes les dépendances et ne peuvent

qui nous intéresse est donc la parallélisation d'un nid de boucles par décalage d'instructions. Nous prouvons que ce problème est NP-complet au sens fort et nous proposons une méthode de résolution par un système de contraintes linéaires entières qui peut être vu comme une base à l'intégration d'un décalage contrôlé dans les algorithmes classiques. Nous identifions également des cas solubles en temps polynomial que nous pensons fréquents en pratique (entre autres, tous les cas de graphes acycliques sont solubles polynomialement).

Au chapitre 5, nous tâchons de répondre à une question sur le décalage ayant trait à la localité des dépendances entre opérations. Ce problème est l'inverse de celui étudié dans le chapitre 2[6], il est intéressant d'un point de vue purement algorithmique afin de mieux comprendre de façon générale les mécanismes impliqués dans les problèmes de décalage d'instructions. Là encore, nous prouvons que le problème est NP-complet au sens fort et nous proposons une méthode de résolution à base de programmation linéaire. Ce problème peut se poser de façon pratique dans les cas où la localité ou la non localité d'une dépendance est plus importante que sa distance (en synthèse de circuits, cela reviendrait à un fil à la place d'un « FIFO », en alignement il s'agit d'éviter la latence des communications).

Enfin, il semble pertinent de souligner l'évolution récente de la compilation de circuits pour systèmes embarqués. Le but de ce domaine de recherche est de produire de façon automatique un circuit sur silicium dont le développement manuel serait coûteux en moyens humains et en temps. Comme son nom l'indique, ce domaine est une variante de la compilation classique : comme celle-ci, elle met en œuvre des transformations de programmes suivies d'optimisations de bas niveau afin de créer un circuit à la fois efficace et peu coûteux à produire. De par leur généralité, nos résultats, initialement conçus pour l'optimisation de programme pour le calcul haute performance, peuvent trouver leur place dans ce contexte de compilation de circuits. À l'exemple du compilateur PICO développé aux HP Labs ([98]), qui est un compilateur de circuits amené à réaliser des transformations de nids de boucles (pour créer une architecture de type « pseudo-systolique »), du pipeline logiciel (pour créer un processeur VLIW spécifique) et de l'optimisation mémoire (pour diminuer le coût en registres et en mémoire tampon de l'architecture).

Par la suite, nous traitons l'ensemble de ces problèmes de compilation comme des problèmes de graphe. Avant de passer au décalage d'instructions proprement dit, il convient donc de présenter quelques préliminaires précisant l'origine de notre représentation et sa justification, ainsi que les bases du décalage d'instructions. C'est ce que nous faisons au chapitre suivant (le chapitre 1).

---

donc pas obtenir de nids de boucles parallèles dont le corps est séquentiel.

[6] Dans le chapitre 2, nous cherchons à éliminer les dépendances séquentielles ou « alignées », alors qu'ici nous cherchons au contraire à les « aligner » au maximum.

Chapitre 1

Préliminaires

1.1 Définitions, modèle

Dans cette section, nous présentons un grand nombre de définitions relatives au modèle que nous utilisons. Nous ne nous servirons pas par la suite de certaine d'entre elles, leur rôle ici étant essentiellement de rappeler d'où vient la représentation d'un programme par un graphe de dépendance et quelle correspondance il existe entre un programme réel et un tel graphe.

1.1.1 Boucles et boucles imbriquées

L'ensemble des résultats présentés tout au long de cette thèse est relatif à l'étude des boucles d'un programme. Intuitivement une boucle est une structure de programme exécutant de façon répétée la même suite d'instructions. Une *instruction* est une structure de programme finie pouvant être une affectation, l'évaluation d'une expression mathématique, un appel de procédure ou une structure de contrôle. Cette définition est essentiellement intuitive, et nous resterons très informels sur la notion d'instruction et sur la structure syntaxique d'un programme[1]. Nous admettrons qu'une instruction complexe comme une structure de contrôle (boucle, test, etc.) puisse contenir d'autres instructions elles-mêmes simples ou complexes. Certaines instructions, les boucles, sont d'un intérêt particulier pour nous.

Définition 1 (boucle) *Une boucle est une instruction de la forme*[2] *:*

Do $i = L, U, S$
  $I_1(i)$
  $\vdots$
  $I_n(i)$
Enddo

où L, U et $S \neq 0$ sont des expressions arithmétiques entières que nous appelons respectivement borne inférieure, borne supérieure et pas de la boucle. $B = \{I_1, \ldots, I_n\}$ est un ensemble ordonné d'instructions que nous appelons corps de la boucle. i est une variable que nous appelons compteur ou indice de boucle. Une boucle est entièrement définie par la donnée du quintuplet (i, L, U, S, B).

Par la suite, nous nous limiterons toujours au cas où une instruction ne change pas le flot de contrôle de la boucle[3]. Dans ce cas, une boucle correspond à la répétition de la séquence d'instructions

[1] Pour une introduction à l'analyse lexicale et syntaxique proprement dite voir [1].
[2] Nous avons gardé les termes anglais dans cette définition afin de rester proches du langage FORTRAN.
[3] Pas de « goto » ni d'instruction changeant la valeur de l'indice de boucle.

formée par son corps pour des valeurs de i variant entre L et U par pas de S. Chaque répétition de la séquence d'instructions est une itération, et sémantiquement une itération est équivalente à l'exécution de chacune des instructions du corps dans l'ordre défini sur $\{I_1, \ldots, I_n\}$, ou ordre textuel. Nous ne supposons aucune forme particulière pour les expression L, U et S, excepté qu'elles doivent correspondre à des valeurs entières[4]. L'étude des transformations de boucles va nous conduire à considérer l'ensemble des valeurs prises par i, ou domaine d'itération.

Définition 2 (domaine d'itération) *Soit une boucle $l = (i, L, U, S, B)$ et la suite $\forall n \in \mathbb{N}, V_n = L + nS$. Le domaine d'itération de l est l'ensemble :*

$$D_l = \left\{ i \in \mathbb{Z} | \exists n \in \mathbb{N}, i = V_n \wedge \left\{ \begin{array}{l} L \leq i \leq U \text{ si } S > 0 \\ U \leq i \leq L \text{ sinon} \end{array} \right. \right\}$$

Les élément de D_l sont les indices d'itération de l. Tout couple $(I, i) \in B \times D_l$ est une opération. L'ordre sur D_l défini par la suite V_n est l'ordre séquentiel de la boucle.

Cette définition du domaine d'itération d'une boucle reprend la sémantique des boucles du langage FORTRAN qui convient parfaitement au cadre de cette thèse. Par la suite nous supposerons que toutes nos boucles ont un pas et une borne inférieure égaux à 1[5]. Dans ce cadre restreint, l'ordre séquentiel d'une boucle est tout simplement l'ordre sur les entiers naturels.

Une boucle contenant une autre boucle forme un ensemble de boucles imbriquées également appelé nid de boucles. Un nid de boucles parfait N peut être défini récursivement comme une boucle contenant soit un nid de boucles parfait soit une séquence d'instructions. Le corps du nid N

```
Do i = 1, n
   Do j = 1, m
      S1 : a(i, j) = b(i − 1, j + 1)
      S2 : b(i, j) = a(i, j − 1)
   Enddo
Enddo
```

FIG. 1.1 – Un exemple de nid de boucles parfait.

est le corps de la boucle la plus interne (la séquence d'instructions), sa profondeur p est le nombre de boucles le définissant, son domaine d'itération est le produit $D_N = D_{l_1} \times \cdots \times D_{l_p}$ des domaines des boucles $\{l_1, \ldots, l_p\}$ qui le composent, et la dimension de son domaine d'itération $dim(D_N)$ est égale à la profondeur du nid p. La figure 1.1 présente un exemple de nid de boucles parfait. Un élément de D_N est un vecteur d'itération de N (généralisation de l'indice pour une simple boucle), une opération de N est un couple formé par une instruction du corps de N et un élément de D_N, l'ordre séquentiel sur D_N est défini par l'ordre lexicographique sur les entiers. Par la suite, nous noterons v_x la $x^{\text{ième}}$ composante d'un vecteur v, l'ordre lexicographique se définit alors comme suit :

Définition 3 (ordre lexicographique) *L'ordre lexicographique sur $\mathbb{Z}^p$ est la relation $<_{lex}$ définie par :*

$$\forall i, j \in D_N, i <_{lex} j \iff \exists k \leq p, (\forall x < k, i_x = j_x) \wedge (i_k < j_k)$$

[4]Cependant, l'analyse de dépendances ainsi que d'autres travaux ne traitent généralement que les cas de fonctions affines de l'indice de boucle, voir par exemple [25, 41, 42].

[5]Si ce n'est pas le cas pour une boucle l, nous pouvons construire une nouvelle boucle équivalente en remplaçant dans le corps de l l'indice i par l'expression $L + (i-1)S$ et en faisant varier l'indice de la nouvelle boucle entre 1 et $|D_l|$ par pas de 1.

Le cas des nids de boucles parfaits est un cas particulier puisque chaque boucle excepté la dernière contient exactement une boucle dans son corps. Un nid de boucles non parfait est une généralisation au cas d'une imbrication quelconque des boucles, il est défini comme une séquence d'instructions. L'ensemble des boucles entourant une instruction I appartenant à un nid non parfait est le nid parfait englobant de I. Le domaine d'itération D_I d'une instruction I (appartenant à un nid non parfait), et sa profondeur sont ceux de son nid englobant.

Toutes ces définitions, bien que parfois intuitives, formalisent néanmoins une notion fondamentale dans toutes nos définitions : la notion de domaine d'itération. L'ensemble des travaux présentés dans cette thèse joue sur le déplacement des instructions dans leur domaine d'itération. De plus la notion classique de parallélisme dans les boucles se fonde sur le découpage du domaine d'itération en sous-ensembles « indépendants ». Il nous faut donc préciser ce que nous entendons par indépendants.

1.1.2 Notion de dépendance

Traditionnellement, un programme est une suite d'opérations exécutées complètement et de manière atomique les unes après les autres de façon totalement séquentielle. Une telle définition découle naturellement de la sémantique des langages de programmation les plus courants (et le FORTRAN n'est pas une exception), elle implique la présence d'un ordre total sur les opérations qui composent le programme. Dans le cas des boucles, cet ordre total est défini par l'ordre séquentiel sur les itérations et l'ordre textuel des instructions de son corps. Si nous prenons l'exemple d'un compilateur paralléliseur, la notion même de parallélisme va à l'encontre de cette vision d'un programme puisque le but même d'un programme parallèle est d'exécuter plusieurs opérations simultanément. De façon plus générale, la démarche de transformation d'un programme séquentiel est donc d'emblée confrontée à un problème vis-à-vis du programme source : le but est de produire un programme différent – donc ne respectant pas nécessairement d'ordre total sur les opérations, ce sera le cas d'un programme parallèle – ayant le même comportement que le programme original – donc respectant toutefois une partie de l'ordre total induit par le programme séquentiel, celle assurant la sémantique. Toute transformation d'un programme séquentiel visant à changer l'ordre d'exécution des opérations doit donc préalablement identifier un ordre partiel sur celles-ci permettant de conserver la sémantique du code original. Cette recherche d'une séquentialité irréductible à l'intérieur du programme est l'analyse de dépendance. Le problème général de l'analyse de dépendance d'un programme sort du cadre de cette thèse, et nous invitons le lecteur intéressé à se reporter à [40, 15, 7, 17] pour une introduction aux différents aspects de ce domaine. Nous rappelons néanmoins ici les principes sous-jacents à l'analyse de dépendances et les moyens pratiques d'expression des dépendances que nous utiliserons par la suite.

Deux opérations sont dites dépendantes si le résultat de leur exécution dépend de l'ordre dans lequel elles sont exécutées. Classiquement, un programme produit un résultat sous la forme d'un contenu particulier de la mémoire (tous les types d'entrées/sorties n'étant que des déclinaisons de ce principe), et les opérations n'ont un effet que dans la mesure où elles accèdent à la mémoire. Il en découle une condition nécessaire à la dépendance de deux opérations, énoncée par Bernstein [8] : deux opérations dépendantes accèdent toutes deux au même emplacement mémoire, et l'un au moins de ces accès est un accès en écriture. Cette condition n'est pas suffisante, l'exemple simple de deux opérations écrivant la même valeur dans une même case mémoire illustre le fait que deux opérations remplissant la condition peuvent être indépendantes. Cependant cette condition est généralement employée comme critère de dépendance lors de l'analyse afin de produire un ensemble de dépendances non pas exact mais conservatif, c'est-à-dire contenant au moins toutes les dépendances réelles du

programme. De plus, cette condition peut souvent être considérée comme la seule utilisable en pratique. En effet, une analyse plus fine permettant de repérer les autres cas d'indépendance requiert généralement une connaissance sémantique du programme et/ou une connaissance du contenu de la mémoire lors de l'exécution.

La nature des accès mémoires provoquant une dépendance détermine la nature de la dépendance elle-même. Quatre cas de figure se présentent :

- une écriture suivie d'une lecture : la dépendance est dite dépendance de flot ou vraie dépendance ;
- une lecture suivie d'une écriture : la dépendance est dite anti-dépendance ;
- une écriture suivie d'une écriture : la dépendance est dite dépendance de sortie ;
- une lecture suivie d'une lecture : il n'y a pas de dépendance.

Il s'avère que les seules dépendances réellement inévitables sont les dépendances de flot ou vraies dépendances, elles correspondent à la propagation des données à l'intérieur du programme. Les autres dépendances (dites aussi fausses dépendances) peuvent être éliminées par diverses techniques de transformation du programme (voir [104, 39, 88]), elles sont dues à l'implémentation et correspondent généralement à un emplacement mémoire réutilisé afin de stocker plusieurs résultats temporaires successifs. Cependant il peut être souhaitable d'autoriser toutes les dépendances dans notre représentation d'un problème[6]. L'exemple de la figure 1.1 ne comporte que des dépendances de flot, la figure 1.2 représente les opérations de cet exemple dans leur domaine d'itération, ainsi que les dépendances entre opérations. Sur cette dernière figure, chaque groupe de deux carrés represente les deux opérations de la boucle, le flèches représentent les dépendances entre opérations, et les carrés en pointillés entourent le domaine d'itération respectif de chaque instruction (ils nous aideront à mieux comprendre les transformations).

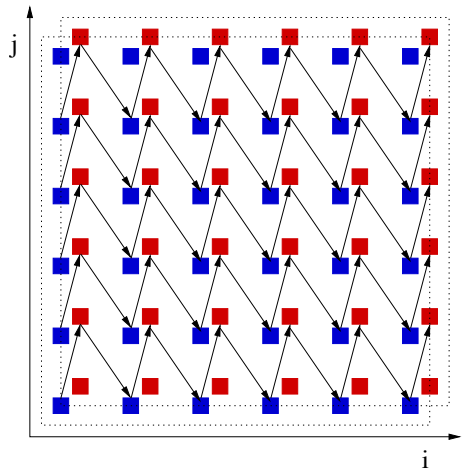


FIG. 1.2 – Dépendances entres opérations de l'exemple 1.1.

Revenons au cas de nos nids de boucles. Afin d'exécuter correctement un nid, les contraintes de dépendance que nous devons respecter sont données par un ordre partiel sur l'ensemble des opérations exécutées par le nid.

Définition 4 (relation de dépendance) *Soient deux instructions I_1 et I_2 (non nécessairement distinctes) ayant pour domaines d'itérations respectifs D_{I_1} et D_{I_2}. Ces instructions correspondent*

[6]Par exemple lors de la transformation d'un morceau de code en langage machine, l'utilisation des registres est alors explicite et un renommage peut s'avérer délicat ou tout simplement non profitable à cause de leur nombre limité.

à deux ensembles d'opérations $O_1 = \{(I_1, i) | i \in D_{I_1}\}$ *et* $O_2 = \{(I_2, i') | i' \in D_{I_2}\}$ *et des dépendances peuvent exister entre éléments de ces deux ensembles. L'ensemble des dépendances de* I_1 *vers* I_2 *est une relation binaire sur* $O_1 \times O_2$ *que nous notons* $\rightarrow$*. Deux opérations dépendantes sont deux opérations* (I_1, i) *et* (I_2, i') *telles que* $(I_1, i) \rightarrow (I_2, i')$*.* (I_1, i) *est l'opération source et* (I_2, i') *l'opération dépendante.*

Bien que cette définition procure un moyen de représenter de façon exacte[7] une relation de dépendance entre les opérations correspondant à deux instructions, son utilisation est peu envisageable en pratique : d'une part une telle représentation impliquerait un trop gros volume de données, le domaine d'une boucle pouvant être grand, et d'autre part peu d'algorithmes seraient à même d'en tirer parti. De nombreuses représentations plus simples pour les dépendances ont donc été proposées, permettant une expression compacte de la relation de dépendance au prix d'une sur-approximation de celle-ci (de façon générale, plus la représentation est compacte, plus la sur-approximation est grossière). Parmi les plus connues nous pourrons trouver les représentations par polyèdres, vecteurs de direction ou niveaux (voir [104] pour une présentation de toutes ces représentations).

La plupart des représentations des dépendances tirent parti de la régularité des boucles en exprimant non pas tous les couples d'opérations dépendantes, mais toutes les distances possibles entre deux opérations dépendantes.

Définition 5 (distance de dépendance) *Soient deux opérations dépendantes* (I_1, i) *et* (I_2, i') *(en d'autres termes* $(I_1, i) \rightarrow (I_2, i')$*), avec* $i = (i_1, \ldots, i_x)$ *et* $i' = (i'_1, \ldots, i'_{x'})$ *deux vecteurs d'itération de même dimension (si ce n'est pas le cas, nous restreignons le plus grand des deux à ses* $\min(x, x')$ *premières composantes). La distance de dépendance entre* (I_1, i) *et* (I_2, i') *est la différence* $i' - i$*.*

Dans le cas de deux opérations dépendantes appartenant à un nid de boucles parfait cette distance traduit directement le nombre d'itérations effectuées sur chaque boucle entre l'exécution de l'opération source et l'exécution de l'opération dépendante. En tant que distance, elle est à interpréter à l'aide de l'ordre lexicographique. Nous la disons légale lorsque sa valeur est lexico-positive ou nulle, et illégale lorsque sa valeur est lexico-négative (dans un nid de boucles parfait, les itérations sont ordonnées par l'ordre lexicographique, ainsi une dépendance lexico-négative signifie que l'opération source est exécutée après l'opération destination. Dans le cas d'une dépendance de flot par exemple, cela revient à utiliser un résultat avant son calcul). La nature répétitive de l'exécution d'une boucle, les accès aux données souvent sous la forme de fonction affine des indices de boucle, et la nature très régulière de nombreux calculs scientifiques permettent souvent de grouper ces distances de dépendances en une représentation plus compacte, sur laquelle les algorithmes de transformation de programme peuvent travailler. Dans cette logique, le cas offrant le plus de régularité, et donc la représentation des dépendances la plus simple, est celui des dépendances uniformes.

Définition 6 (relation de dépendance uniforme) *Soient deux instructions* I_1 *et* I_2 *ayant pour domaines d'itérations respectifs* D_{I_1} *et* D_{I_2} *de même dimension (si ce n'est pas le cas nous restreignons le plus grand des deux à ses* $p = \min(dim(D_{I_1}), dim(D_{I_2}))$ *premières dimensions). La relation de dépendance* $\rightarrow$ *entre* I_1 *et* I_2 *est uniforme si et seulement si :*

$$\exists! x \in \mathbb{Z}^p, \forall (i, i') \in D_{I_1} \times D_{I_2}, (I_1, i) \rightarrow (I_2, i') \Rightarrow i' - i = x$$

Autrement dit, dans le cas d'une relation de dépendance uniforme, la connaissance d'un simple vecteur fournit une sur-approximation minimale[8] d'une relation de dépendance entre deux instructions.

[7] Évidemment la possibilité de représenter un ensemble de dépendances de façon exacte ne rend pas son analyse plus simple. De façon générale, l'analyse de dépendances reste une sur-approximation des dépendances réelles.

[8] L'ensemble des distances de dépendance possibles se réduit à un seul élément.

Définition 7 (dépendances uniformes) *Nous disons que les dépendances entre deux instructions I_1 et I_2 sont uniformes si la relation de dépendance qui leur est associée est l'union d'un nombre fini x de relations de dépendance uniformes où x est indépendant de D_1 et D_2.*

Dans l'ensemble de nos travaux, nous supposerons les dépendances toujours uniformes, et nous représenterons la relation de dépendance entre instructions par l'ensemble de distances approprié.

1.1.3 Graphes de dépendance

Dans tous les problèmes que nous traitons, nous avons besoin d'une représentation globale nous permettant de regrouper l'ensemble des informations qui nous seront utiles afin de manipuler et de transformer une boucle ou un nid de boucles. D'un point de vue général, nous considérons qu'une boucle peut avoir un grand nombre d'itérations et nous souhaitons regrouper les informations relatives aux opérations constituant son corps aussi bien que les informations relatives aux dépendances existantes entre les opérations exécutées durant ses différentes itérations. Une possibilité est l'utilisation d'un graphe de dépendance étendu, contenant un sommet par opération contenue dans chaque itération de la boucle, et un arc par dépendance entre deux opérations. En d'autres termes, la boucle est déroulée et les relations de dépendance exprimées entre instances spécifiques d'opérations, c'est-à-dire de façon exacte. C'est ainsi que nous avions représenté les dépendances entre opérations sur la figure 1.2. Il est d'emblée exclu de travailler sur une telle représentation de la totalité des itérations réalisées par la boucle d'une part parce qu'il est impossible de limiter *a priori* sa taille (taille du domaine d'itération), d'autre part parce que nous souhaitons une solution compacte, dont la taille est du même ordre que celle de la boucle originale, et non de son graphe étendu. Nous nous intéressons donc au graphe de dépendance réduit, que nous appellerons abusivement par la suite graphe de dépendance.

Définition 8 (graphe de dépendance réduit) *Un graphe de dépendance réduit associé à un nid de boucle N de profondeur p est un graphe orienté $G = (V, E)$. Chaque sommet de V est associé à une instruction du corps de N, et chaque arc à une relation de dépendance entre deux sommets (instructions). Les arcs sont pondérés par une fonction d qui précise la relation de dépendance entre deux instructions. Par extension nous appelons p la profondeur du graphe. Nous notons également le graphe de dépendance $G = (V, E, d)$ lorsque nous voulons préciser la pondération d de façon explicite.*

Un graphe de dépendance est donc une représentation générique de la structure d'un nid de boucles. C'est une base qui permet de représenter une information structurelle sur notre boucle ou nid de boucles ; à celle-ci s'ajoute un certain nombre de pondérations des arcs et des sommets permettant de préciser la nature des instructions, les détails de leur exécution ou des informations additionnelles sur les dépendances. Un graphe de dépendance contient toujours une pondération d sur ses arcs qui précise la relation de dépendance associée. De façon abusive nous la désignons par poids d'un arc – le poids se réfère donc par défaut à la pondération d. Chaque sommet correspond à une des instructions génériques du corps du nid, un sommet correspond donc à une instruction au sens textuel du terme, telle qu'elle apparaît dans le code source du programme. De façon abusive nous utilisons de manière identique et interchangeable les notions d'instruction et de sommet d'un graphe de dépendance. Comme présenté dans la définition 2, une instance particulière d'une instruction est donc définie par un couple constitué d'un sommet du graphe et d'un vecteur d'itération i. Nous utilisons les notions classiques de chemin ou chaîne, de circuit ou cycle et de poids d'un chemin, d'une chaîne, d'un circuit ou d'un cycle (voir [38] pour de plus amples détails).

Lorsque nous sommes dans le cas de problèmes à dépendances uniformes, chaque ensemble de dépendances entre deux instructions peut être décrit par un nombre fini de relations de dépendance

uniformes. De plus, chacune de ces relations peut être décrite par un simple vecteur entier (une distance de dépendance). Ainsi un problème en dépendances uniformes pourra être décrit par un multi-graphe dont chacun des arcs décrit l'une des relations de dépendance uniformes.

Définition 9 (graphe de dépendance uniforme) *Un graphe de dépendance uniforme est un graphe de dépendance $G = (V, E, d)$ dont la pondération d associe à chaque arc de E un vecteur entier de dimension inférieure ou égale à la profondeur p de G.*

Ainsi dans un graphe de dépendance uniforme, un arc e entre deux sommets u et v de poids $d(e)$ exprime un ensemble de dépendances des opérations (u, i) vers les opérations $(v, i+d(e))$ où $i \in D_u$. Le graphe de dépendance de l'exemple de la figure 1.1 est représenté sur la figure 1.3, c'est un graphe de dépendance uniforme.

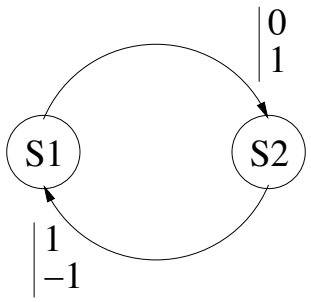


FIG. 1.3 – Graphe de dépendance réduit de l'exemple 1.1.

De prime abord, nous imposons une contrainte au graphe afin de garantir notre capacité à produire le code correspondant : l'absence de circuits de poids lexico-négatif.

Définition 10 (graphe correct) *Un graphe de dépendance est dit correct s'il ne contient aucun circuit c tel que $d(c) \leq_{lex} 0$.*

En effet, un tel circuit signifierait, sous l'hypothèse d'une exécution suivant l'ordre lexicographique, qu'une opération dépend d'une de ses instances exécutée dans les itérations futures. Rappelons que nous considérons l'ordre lexicographique comme l'ordre séquentiel d'un nid de boucles, c'est-à-dire l'ordre de référence définissant la sémantique de notre programme. Il nous est donc impossible de produire un code correct associé à un tel graphe. Le code de départ est par définition le modèle sémantique à respecter, il est donc correct par nature. Cependant, une transformation intermédiaire, comme un échange de boucle par exemple, peut produire un graphe incorrect. Certaines transformations peuvent traiter le cas d'un tel graphe : si tous les circuits ont un poids lexico-négatif par exemple, il est possible grâce à un renversement de boucle[9] de les rendre tous de poids lexico-positif. Néanmoins, comme nous le verrons par la suite, les transformations qui font l'objet de cette thèse ne modifient pas le poids d'un circuit, et ne font pas partie des transformations pouvant corriger un graphe incorrect. Nous devons donc interdire les circuits de poids lexico-négatif, et les transformations additionnelles pouvant éliminer tous ces circuits sortent du cadre de cette étude. Enfin nous définissons la notion de graphe légal.

Définition 11 (graphe légal) *Un graphe légal est un graphe correct $G = (V, E, d)$ tel que :*

$$\forall e \in E, d(e) \geq_{lex} 0$$

[9]Le renversement de boucle consiste à exécuter les itérations dans l'ordre séquentiel décroissant.

Un graphe légal correspond directement à un nid de boucles parfait exprimé dans un langage de haut niveau séquentiel comme, par exemple, le FORTRAN (nous admettrons l'existence d'une telle traduction respectant l'ordre séquentiel).

Maintenant que nous avons défini les bases de notre représentation en termes de dépendances et de code associé, revenons aux pondérations de notre graphe. Dans le cadre du pipeline logiciel, la distance de dépendance entre deux opérations est une information nécessaire mais incomplète. En effet, dans ce cas les opérations sont ordonnancées explicitement dans le temps de manière concurrente : sont importants non seulement la succession des itérations, mais également les cycles de départ et de fin d'une opération au niveau du processeur lui-même. Ainsi, il devient nécessaire de connaître plus de détails sur l'exécution de chaque opération, d'une part pour déterminer un ordonnancement correct des opérations, et d'autre part pour déterminer le nombre de ressources requises à chaque instant pour exécuter l'ordonnancement choisi. Afin de modéliser ces détails sur l'exécution d'une opération (qui dépendent de l'architecture cible), nous utilisons une ou plusieurs pondérations additionnelles selon les deux cas de figure suivants.

Ressources : modèle non pipeliné

Nous décrivons ici un modèle simplifié que nous appelons, par abus de langage, modèle non pipeliné. Dans ce modèle, une opération s'exécute de manière atomique en monopolisant les ressources nécessaires durant toute son exécution et en ne produisant un résultat qu'une fois celle-ci totalement terminée. Le terme de « non pipeliné » ne fait donc pas référence à l'architecture cible, mais se réfère au comportement de l'opération. Une opération dans notre modèle non pipeliné est caractérisée par son temps d'exécution.

Définition 12 (temps d'exécution) *Soit un graphe de dépendance $G = (V, E, d)$. Le temps de calcul ou temps d'exécution d'une instruction $u \in V$, noté $t_c(u)$, est le nombre de cycles (du processeur) nécessaire au calcul d'une instance de u. t_c est une pondération entière des sommets du graphe.*

Ainsi, pour tout couple d'opérations dépendantes, il doit s'écouler un intervalle de temps au moins égal au temps d'exécution de l'opération source entre le début de l'exécution de cette dernière et le début de l'exécution de l'opération dépendante : c'est la latence entre deux opérations.

Ressources : modèle pipeliné

Dans le modèle pipeliné, par opposition au modèle non pipeliné présenté précédemment, l'exécution d'une opération n'est plus atomique. Des ressources peuvent être libérées au cours de celle-ci, et des résultats peuvent être produits avant sa fin. Cela signifie que le temps d'exécution d'une opération ne caractérise plus ni le temps de latence entre deux opérations dépendantes ni le temps de réservation des ressources mises en œuvre pour son exécution. Là encore, ce découpage de l'opération n'est pas nécessairement lié à l'architecture cible, cette dernière pouvant ne pas être pipelinée (en revanche une architecture pipelinée correspond toujours à des opérations libérant des ressources avant la fin de leur exécution). Dans ce modèle, le temps d'exécution d'une opération est défini comme dans la définition 12, mais une information plus précise sur les opérations est requise.

Tout d'abord, nous avons besoin d'une information sur la latence, il s'agit du délai à respecter entre deux opérations dépendantes. Les dépendances provenant des accès au mêmes endroits de la mémoire par des opérations différentes, la latence entre deux opérations dépend donc du type de dépendance qui les relie (nature de l'accès de chaque opération : lecture ou écriture) ainsi que de

l'étage du pipeline dans lequel se produit l'accès (les lectures se produisent généralement au début du pipeline, tandis que les écritures se produisent généralement à la fin du pipeline).

Définition 13 (latence) *Soit un graphe de dépendance $G = (V, E, d)$. La latence d'une dépendance $e = (u, v) \in E$, notée $l(e)$, est le nombre minimum de cycles (processeur) devant séparer le début de l'exécution de (u, i) du début de l'exécution de $(v, i + d(e))$ afin de respecter la dépendance.*

Remarquons que la latence peut prendre *a priori* toutes sortes de valeurs, aussi bien positives que négatives. Par exemple, dans le cas d'une dépendance de flot, il est nécessaire d'attendre l'écriture en fin de pipeline du résultat de l'opération source dans un registre avant de commencer à charger les opérandes de l'opération dépendante dans les premiers étages du pipeline, la latence sera sûrement positive. En revanche, dans le cas d'une anti-dépendance, l'opération dépendante doit juste attendre le chargement des opérandes par l'opération source avant de procéder à l'écriture dans un registre, ceci pouvant même conduire à une latence négative – il suffit juste que l'opération source ait le temps de démarrer et de charger ses opérandes durant l'exécution de l'opération dépendante. Notons toutefois que la plupart de nos résultats théoriques ne tiennent que pour des latences non négatives, nous serons donc souvent conduits à nous limiter à ce cas. Cette limitation n'est pas réellement restrictive puisque des latences négatives sont relativement rares. De plus, elles correspondent à des anti-dépendances ou à des dépendances de sortie. Ces dépendances dites également fausses dépendances peuvent être éliminées par l'emploi de diverses techniques (voir par exemple [104] sur l'élimination des fausses dépendances).

Considérons maintenant le temps d'utilisation d'une ressource. Nous considérons une ressource comme libre à partir du moment où elle peut accueillir une nouvelle opération. Dans le cas d'un processeur pipeliné, il devient alors évident que nous devons séparer cette notion de la notion de temps de calcul : une opération doit traverser tous les étages du pipeline afin de terminer son calcul, mais une nouvelle opération peut démarrer dès que le premier étage est libéré.

Définition 14 (temps de réservation d'une ressource) *Soit un graphe de dépendance $G = (V, E, d)$. Le temps de réservation d'une ressource par une instruction u, noté $t_r(u)$, est le nombre de cycles processeur nécessaires à partir du début du calcul d'une instance de u avant de pouvoir démarrer le calcul d'une autre opération sur la même ressource.*

Cette représentation de l'utilisation des ressources est extrêmement simpliste et ne convient qu'au cas où les instructions n'utilisent qu'une seule ressource pour leur calcul. Afin de modéliser des cas plus complexes (par exemple si une instruction accède à la fois à l'ALU du processeur et au bus mémoire durant son calcul), nous pouvons utiliser des tables de réservation (voir [67, 93]) indiquant quelles ressources sont utilisées à chaque cycle lors de l'exécution de l'opération. Pour l'ensemble de nos résultats théoriques, nous nous limiterons au cas de la définition 14, et nous utiliserons une représentation par tables de réservation lors de la mise en œuvre d'expérimentations.

Considérations sur les ressources

Revenons tout d'abord au cas du modèle de ressources non pipelinées. Dans ce cas, à cause de l'atomicité des opérations, le temps d'exécution d'une opération, son temps de réservation des ressources et la latence vers les opérations qui en dépendent sont égaux. C'est pourquoi nous n'avons eu besoin de modéliser que le temps d'exécution d'une opération pour décrire son comportement. En définitive, ce cas n'est qu'un cas particulier du modèle pipeliné dans lequel pour tout arc $e = (u, v)$, nous avons $t_c(u) = t_r(u) = l(e)$. Par la suite, afin d'uniformiser les notations, nous utiliserons toujours le modèle pipeliné soumis à la condition précédente pour décrire la situation dans le modèle non pipeliné. Évidemment, la contrainte d'égalité des trois valeurs de ce modèle ne nous permettra pas de démontrer les mêmes choses et nous serons tout de même conduits à séparer les deux cas.

1.2 Transformations de nids de boucles

Dans cette thèse nous souhaitons étudier certaines transformations de nids de boucles destinées à produire de façon automatique un code optimisé. Il semble important de préciser que par transformations nous entendons principalement réordonnancement des opérations. Nous ne considérons pas les autres optimisations que nous considérons comme de bas niveau (dans la chaîne de compilation). Ces autres optimisations peuvent consister en une modification des opérations exécutées par le programme (par exemple remplacer une multiplication par 2 par un décalage), en l'alignement des données en mémoire, etc. Toutes ces autres techniques sortant du cadre de ce travail, nous nous limiterons essentiellement au cas du réordonnancement des instructions.

Avant de présenter le décalage d'instructions qui est notre principal objet d'étude, nous présentons une autre transformation classique de nids de boucles, la torsion de boucle.

1.2.1 Un exemple de transformation : la torsion de boucles

Dans notre définition d'un nid de boucles et de l'ordre séquentiel, le vecteur d'itération parcourt le domaine d'itération suivant l'ordre lexicographique. Autrement dit, sa valeur varie suivant les vecteurs de la base canonique du domaine d'itération (dans chaque dimension la valeur augmente du pas de la boucle à la manière du compteur kilométrique d'une voiture). La torsion de boucles (*loop skewing*) consiste à changer cet ordre de parcours du domaine d'itération simplement en effectuant un changement de base (à vrai dire, la torsion de boucles est un peu moins générale que ça, elle s'applique à deux boucles et utilise un changement de base particulier défini par son facteur, voir [110, 109]). Une façon alternative de voir cette technique est de considérer que le domaine d'itération de notre nid de boucles est déformé – et donc les opérations exécutées dans un ordre différent – pour prendre la forme d'un parallélogramme. La figure 1.4 est un exemple de torsion de boucles appliquée à l'exemple de la figure 1.2.

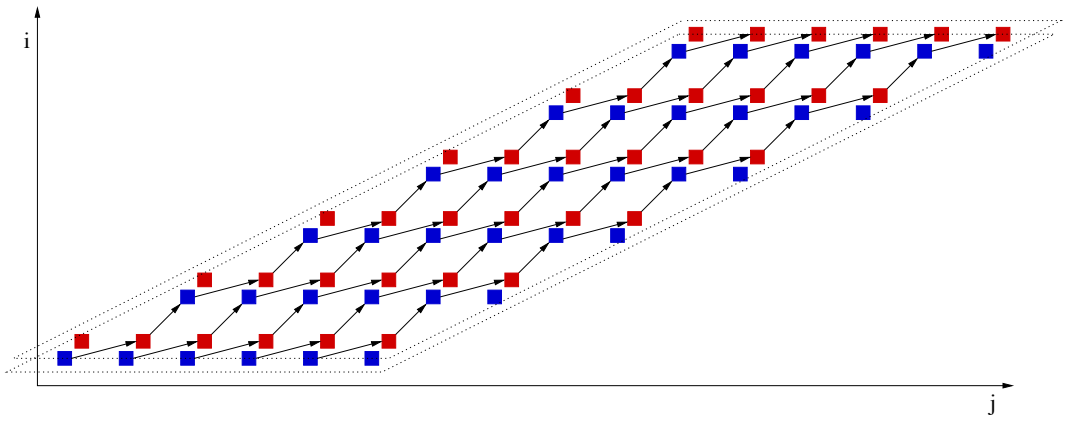


FIG. 1.4 – L'exemple 1.1 après une torsion de boucles.

1.2.2 Le décalage d'instructions (*retiming*)

Le décalage d'instructions est à l'origine une technique d'optimisation connue dans le domaine de la synthèse de circuits (ou VLSI pour *Very Large Scale Integration*) sous le nom de *retiming*. Cette technique a été introduite en 1991 par Leiserson et Saxe (voir [70]) comme un outil théorique visant à minimiser la période d'horloge ainsi que le nombre de registres des circuits synchrones,

modélisés sous forme de graphes dirigés pondérés. Son utilisation s'est par la suite répandue dans le domaine du parallélisme principalement dans le cadre du pipeline logiciel, mais également dans d'autres applications ([5, 10, 13, 26]). La notion de *retiming* dans le cadre de la transformation de code est liée à la notion de déplacement des opérations d'un nid de boucle dans leur domaine d'itération. Appliquer un *retiming* à une instruction d'un nid de boucles revient à appliquer une translation à son domaine d'itération. La notion de temps dans une boucle étant intimement liée à l'ordre lexicographique sur son domaine d'itération, appliquer un *retiming* à une instruction revient donc également à déplacer ses instances dans le temps.

Plus formellement, un *retiming* est une fonction $r : V \to \mathbb{Z}^p$ qui associe à chaque sommet d'un graphe de dépendance, donc à chaque instruction d'un nid de boucles, un vecteur entier de la profondeur de l'instruction correspondante. Cette valeur de *retiming* correspond à un retard que nous souhaitons appliquer à l'exécution de toutes les opérations associées au sommet concerné. Ce retard est exprimé en nombre d'itérations par lequel nous souhaitons reporter l'exécution des opérations (nombre d'itérations sur chaque boucle du nid, à interpréter grâce à l'ordre lexicographique). Appliquer un *retiming* r au graphe $G = (V, E)$ signifie que chaque opération (u, i) correspondant à l'instruction u dont la valeur de *retiming* est $r(u)$ n'est plus exécutée à l'itération i, mais à l'itération $i+r(u)$[10]. Ainsi, une fois le *retiming* appliqué, chaque instruction u de domaine $D_u = \{1, \ldots, n\}$ voit

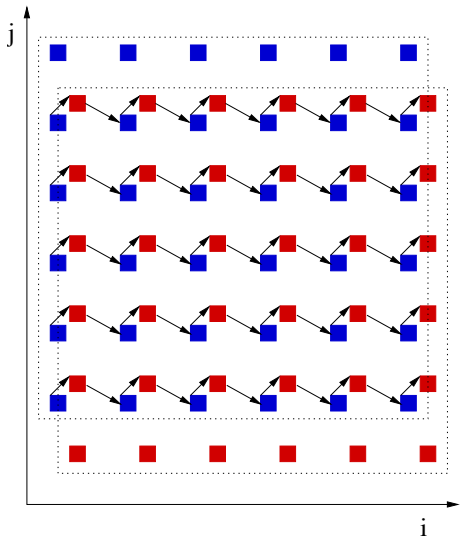


FIG. 1.5 – L'exemple 1.1 après un *retiming*.

son domaine changé en $D'_u = \{1+r(u), \ldots, n+r(u)\}$ obtenu par simple translation de vecteur $r(u)$, d'où le terme de décalage d'instructions que nous utilisons également pour décrire un *retiming*. La donnée de ce *retiming* r ajoute donc une dimension à notre représentation du problème : alors que précédemment la donnée d'une instruction I et d'un vecteur d'itération $i \in D_I$ suffisait à connaître une instance d'instruction $I(i)$ et l'itération i à laquelle elle s'exécutait, il faut désormais connaître également une valeur de *retiming* $r(I)$ afin de déterminer à la fois la même instance d'instruction $I(i)$ et l'itération $i + r(I)$ à laquelle elle s'exécute. La figure 1.5 est un exemple de *retiming* appliqué à l'exemple de la figure 1.2. Sur cette figure, les valeurs de *retiming* sont $r(S1) = (0, 1)$ et $r(S2) = (0, 0)$.

[10] Évidemment tout ceci ne s'applique qu'au cas où la boucle possède un pas égal à 1, ce qui est une de nos hypothèses (voir section 1.1.1). Notons que dans le cas d'un pas négatif l'itération $i + r(u)$ ne correspond plus à un retard mais à une avance.

L'application d'un *retiming* r à une instruction u la fait s'approcher d'autant d'itérations de ses successeurs et s'éloigner d'autant d'itérations de ses prédécesseurs. En d'autres termes, après l'application d'un *retiming* r, deux opérations (u,i) et (v,i') telles que $(u,i) \rightarrow (v,i')$ ne s'exécutent plus respectivement aux itérations i et i', mais respectivement aux itérations $i + r(u)$ et $i' + r(v)$. Ainsi la distance de dépendance entre ces deux opérations devient :

$$\begin{aligned} d_r(u,v) &= (i' + r(v)) - (i + r(u)) \\ &= i' - i + r(v) - r(u) \\ &= d(u,v) + r(v) - r(u) \end{aligned}$$

Définition 15 (*retiming* d'un graphe) *Soit un graphe de dépendance $G = (V,E,d)$ de profondeur p. Une fonction $r : V \rightarrow \mathbb{Z}^p$ est appelée un* retiming *de G. Le graphe $G_r = (V,E,d_r)$ où $\forall e = (u,v) \in E, d_r(e) = d(e) + r(v) - r(u)$ est appelé graphe après* retiming *de G et est noté G_r.*

En termes de graphe, la modification de la distance de dépendance entre deux instructions peut être interprétée comme un « déplacement du poids des arcs à travers un sommet ». Cette manière très intuitive de voir le *retiming* permet de mieux comprendre l'effet de cette transformation sur un graphe de dépendance. La figure 1.6 représente ce déplacement du poids des arcs : sur cette figure chaque unité de poids est représentée par un bâtonnet. Comme nous pouvons le constater, un *retiming* de +1 appliqué sur un sommet enlève une unité de poids de tous ses arcs sortant pour l'ajouter à tous ses arcs entrants. C'est sous cette forme intuitive que le *retiming* est présenté par

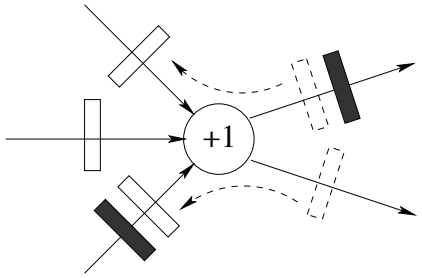


FIG. 1.6 – Déplacement du poids produit par le *retiming*.

Leiserson et Saxe dans [70] ; rappelons toutefois que, dans leur présentation, le *retiming* n'est pas une transformation de programme mais bel et bien un mouvement de registres (le poids des arcs) à travers des unités fonctionnelles (les sommets). Dans ce même article, Leiserson et Saxe montrent la propriété suivante, concernant le poids d'un chemin après *retiming* :

Lemme 1 (Leiserson et Saxe) *Soit $G = (V,E,d)$ un graphe de dépendance, P l'ensemble des chemins de G, et r un* retiming *de G, alors*

$$\forall p = \{u_1, \ldots, u_n\} \in P, d_r(p) = d(p) + r(u_n) - r(u_1)$$

Preuve La preuve est dans [70], elle est immédiate :

$$\begin{aligned} d_r(p) &= \sum_{e=(u,v)\in p} d_r(e) = \sum_{e=(u,v)\in p} (d(e) + r(v) - r(u)) \\ &= \sum_{e=(u,v)\in p} d(e) + r(u_n) - r(u_1) = d(p) + r(u_n) - r(u_1) \end{aligned}$$

□

Corollaire 1 *Soit $G = (V, E, d)$ un graphe de dépendance, C l'ensemble des circuits de G, et r un* retiming *de G, alors*

$$\forall c \in C, d_r(c) = d(c)$$

Le *retiming* possède également une propriété intéressante relative aux cycles d'un graphe. Un cycle d'un graphe G est un chemin non dirigé (dont les arcs peuvent être suivis dans les deux directions) d'un sommet vers lui-même (voir [38]). Si tous les arcs sont suivis dans le même sens, alors le cycle est un circuit, un chemin dirigé d'un sommet vers lui-même. Un cycle (resp. circuit) élémentaire est un cycle (resp. circuit) ne contenant pas deux fois le même sommet. Un cycle élémentaire c, étant donné un sens de parcours, peut être représenté par une fonction $\mu_c : E \longrightarrow \{-1, 0, +1\}$ définie de la manière suivante :

$$\forall e \in E,\ \mu(e) = \begin{cases} +1 & \text{si } e \text{ est un arc de } c \text{ dans le sens du parcours} \\ -1 & \text{si } e \text{ est un arc de } c \text{ opposé au sens du parcours} \\ 0 & \text{sinon (si } e \text{ n'appartient pas à } c) \end{cases}$$

Notons que la connaissance d'une telle fonction nous renseigne simultanément sur les arcs du cycle et sur son sens de parcours. Nous définissons le poids d'un cycle c comme :

$$d(c) = \sum_{e \in c} \mu_c(e) d(e)$$

Nous pouvons alors prouver qu'un *retiming* de G ne change pas le poids de ses cycles (cette preuve peut être trouvée dans [23]) :

Proposition 1 *Étant donné un cycle c d'un graphe G, $d(c) = d_r(c)$ pour tout* retiming *r de G.*

Preuve Sans perte de généralité nous pouvons supposer que c est élémentaire, autrement il se décompose en une union de cycles élémentaires et la preuve montre que le poids de chacun d'eux est inchangé.

$$\begin{aligned} d_r(c) &= \sum_{e \in c} \mu_c(e) d_r(e) = \sum_{e=(u,v) \in c} \mu_c(e)(d(e) + r(v) - r(u)) \\ &= \sum_{e \in c} \mu_c(e) d(e) + \sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u)) \end{aligned}$$

Puisque c est élémentaire, il passe au plus une seule fois par un sommet, donc tout sommet x de c est partagé exactement par deux arcs e_1 et e_2 par lesquels passe le cycle. Donc x apparaît exactement deux fois dans la somme $\sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u))$, et deux cas peuvent se produire :

- e_1 et e_2 sont parcourus dans le même sens, dans ce cas $\mu_c(e_1) = \mu_c(e_2)$ et x apparaît une fois en tant que $r(x)$ et une seconde en tant que $-r(x)$;
- e_1 et e_2 sont parcourus dans des sens opposés, dans ce cas $\mu_c(e_1) = -\mu_c(e_2)$ et x apparaît soit deux fois en tant que $r(x)$, soit deux fois en tant que $-r(x)$.

Dans les deux cas la somme des deux occurrences est nulle, il en résulte que :

$$\sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u)) = 0$$

donc $d_r(c) = d(c)$. □

Intéressons-nous maintenant rapidement au programme obtenu après application d'un *retiming* sur son graphe de dépendance. La génération du programme associé à un tel graphe après *retiming* est relativement simple. Soit $G = (V, E, d)$ un graphe de dépendance associé à un nid de boucles N, si N est parfait il sera de la forme :

Do $i_1 = L_1, U_1$
  $\vdots$
  Do $i_p = L_p, U_p$
    $I_1(i_1, \ldots, i_p)$
    $\vdots$
    $I_n(i_1, \ldots, i_p)$
  Enddo
  $\vdots$
Enddo

Pour tout *retiming* de G nous définissons une traduction générique du code après *retiming* de la manière suivante :

Définition 16 (traduction générique d'un *retiming*) *Soit $G = (V, E)$, un graphe de dépendance associé à un nid de boucles parfait N de profondeur p : N est composé de p boucles imbriquées de la forme $(i_k, L_k, U_k, 1, B_k)$ pour $1 \leq k \leq p$. Soit un* retiming *r de G, $\Delta(r)_k = \max_{I \in B_N} r(I)_k$ et $\delta(r)_k = \min_{I \in B_N} r(I)_k$ respectivement la valeur maximale et la valeur minimale de* retiming *sur la $k^{\text{ième}}$ composante. Le programme suivante est la traduction générique (ou code générique) de r :*

Do $i_1 = L_1 + \delta(r)_1, U_1 + \Delta(r)_1$
  $\vdots$
  Do $i_p = L_p + \delta(r)_p, U_p + \Delta(r)_p$
    if $((i_1, \ldots, i_p) - r(I_{k_1}) \in D_{I_{k_1}})$ then
      $I_{k_1}((i_1, \ldots, i_p) - r(I_{k_1}))$
    $\vdots$
    if $((i_1, \ldots, i_p) - r(I_{k_n}) \in D_{I_{k_n}})$ then
      $I_{k_n}((i_1, \ldots, i_p) - r(I_{k_n}))$
  Enddo
  $\vdots$
Enddo

où $\{I_{k_1}, \ldots, I_{k_n}\}$ est une extension linéaire du graphe $G' = (V, E', d)$ déduit de G en posant $E' = \{e \in E | d_r(e) = 0\}$ (sous-graphe des arcs de poids nul).

Bien sûr, rien ne justifie *a priori* la correction de cette traduction. En appliquant un *retiming*, nous changeons l'ordre d'exécution des opérations du nid de boucles, et en raison de la présence de dépendances, ceci est susceptible de produire un code incorrect (c'est-à-dire ayant une sémantique différente de celle du programme original). Autrement dit, le code générique de la définition 16 n'est correct que si l'ordre qu'il induit sur les opérations respecte l'ordre partiel induit par les dépendances présentes dans le code original. Nous devons donc nous assurer que la variation des distances de dépendances après *retiming* ne compromet pas cette correction.

Tout d'abord, tous les arcs de G tels que $d_r(e) = 0$ expriment des dépendances entre opérations du corps dans une même itération du nid de boucles. Deux instructions reliées par une dépendance de poids nul doivent donc se suivre dans l'ordre textuel. C'est pourquoi nous procédons au réordonnancement $\{I_{k_1}, \ldots, I_{k_n}\}$ du corps du nid de boucles. Ce réordonnancement est toujours possible :

en effet, G est correct (donc il ne contient pas de circuit de poids nul) et r ne change pas le poids des circuits (car c'est un *retiming*), donc G_r ne contient aucun circuit de poids nul et G' est acyclique.

Ensuite, sachant que les itérations de notre nouveau nid sont exécutées suivant l'ordre lexicographique et que la traduction générique de r est un nid de boucles parfait, nous concluons que toutes les distances de dépendances doivent être légales (lexico-positives ou nulles), autrement dit G_r doit être légal. Cette contrainte de légalité sur G_r se dérive en contrainte sur r et nous conduit à définir la notion de *retiming* légal :

Définition 17 (*retiming* légal) *Soit un graphe de dépendance $G = (V, E, d)$ et un* retiming *r de G, nous disons que r est légal si et seulement si :*

$$\forall e \in E, d_r(e) \geq_{lex} 0$$

Ainsi, seuls les codes génériques provenant d'un *retiming* légal sont corrects (notons que cette définition du *retiming* légal est la même que celle proposée par Leiserson et Saxe). Nous admettrons que cette traduction générique d'un *retiming* se généralise aux nids de boucles non parfaits, soit en calculant la constante à ajouter aux bornes de chaque boucle à partir des instructions qu'elle contient et en laissant la structure d'imbrication inchangée, soit en fusionnant totalement le nid en un nid parfait (dans ce cas, la légalité de la fusion est assurée par la légalité du *retiming*).

Évidemment, cette traduction générique peut être simplifiée par l'application de nombreuses optimisations. Parmi celles-ci nous pouvons citer l'épluchage de boucle (*loop peeling*, voir [109, 110]) qui permet de détacher les itérations ne faisant pas partie de l'intersection des domaines de toutes les instructions. Il en résulte un programme qui comporte un prologue et un épilogue (pour chaque boucle), mais duquel les instructions conditionnelles ajoutées à l'intérieur du corps peuvent être éliminées. L'épluchage de boucle peut être particulièrement intéressant pour les boucles ayant un grand domaine d'itération et de relativement faibles valeurs de *retiming*. En particulier, ces formes à prologue et épilogue sont très répandues dans la cadre de l'utilisation de *retiming* pour le pipeline logiciel. Nous ne détaillerons pas plus avant le problème de la génération efficace de code après *retiming*, et nous invitons le lecteur intéressé à se référer à des travaux plus généraux englobant le *retiming* comme cas particulier ([91]). Par la suite nous considérerons qu'étant donné un graphe de dépendance et un *retiming*, le programme correspondant peut toujours s'écrire au moins sous la forme donnée par la définition 16. Nous ne nous intéresserons donc qu'aux problèmes de recherche du *retiming*, en d'autres termes nous nous limiterons à des problèmes de graphe.

Enfin, nous avons défini la notion de *retiming* légal (définition 17), et nous souhaitons généraliser cette notion à la notion de *retiming* sous contrainte : nous disons qu'un *retiming* r d'un graphe G respecte la contrainte de poids minimal m si $\forall e \in E, d_r(e) \geq_{lex} m(e)$. Cette contrainte peut alors être transformée en contrainte de légalité en nous aidant du graphe contraint :

Définition 18 (Graphe contraint par m (ou graphe $G - m$)) *Soit un graphe $G = (V, E, d)$ de profondeur p et une fonction $m : E \to \mathbb{Z}^p$, nous notons $G - m$ (graphe contraint par m) le graphe $G' = (V, E, d')$ où $d'(e) = d(e) - m(e)$.*

Ainsi, r respecte la contrainte de poids minimal m pour G si r est légal pour $G - m$.

Nous pouvons alors présenter un algorithme permettant de trouver un *retiming* r d'un graphe G respectant la contrainte de poids minimal m ou de montrer qu'un tel *retiming* n'existe pas. Par la suite, et dans toute cette thèse, nous serons très souvent amenés à utiliser cet algorithme, c'est pourquoi nous le présentons dès à présent. Cet algorithme utilise l'algorithme de Bellman-Ford pour calculer la longueur du plus court chemin d'un sommet s vers tous les autres, ou bien pour montrer que le graphe contient un circuit de poids négatif. L'utilisation de vecteurs et de l'ordre lexicographique ne change rien à cet algorithme et nous pouvons donc l'utiliser même pour des problèmes multi-dimensionnels.

Algorithme 1 (*Algorithme de Bellman-Ford*)

- pour chaque sommet $v \in V$ définir $\pi(v) = +\infty$;
- pour la source s des chemins cherchés définir $\pi(s) = 0$;
- répéter $|V| - 1$ fois :
  - pour tout arc $e = (u, v) \in E$, si $\pi(u) > \pi(v) + d(e)$ alors $\pi(u) = \pi(v) + d(e)$;
- s'il existe $e = (u, v) \in E$ tel que $\pi(u) > \pi(v) + d(e)$:
  - alors G contient un circuit de poids négatif ;
  - sinon π est la longueur cherchée.

Preuve Voir le livre de Cormen, Leiserson et Rivest [18]. □

Nous pouvons alors présenter notre algorithme de *retiming* sous contrainte, qui trouve un *retiming* r de G qui respecte la contrainte de poids minimal m ou conclut qu'il n'existe pas de tel *retiming* :

Algorithme 2 (retiming *sous contrainte*)

- partir de $G' = G - m$;
- ajouter un sommet s à G' et un arc de poids nul de s vers chacun des autres sommets de G' ;
- appliquer l'algorithme de Bellman-Ford pour trouver un plus court chemin de s vers chacun des autres sommets du graphe. Deux possibilités se présentent :
  - il existe un plus court chemin p_u de poids $\pi(u)$ de s vers chaque sommet u de G', dans ce cas $r = -\pi(u)$ est le *retiming* cherché ;
  - il existe un circuit de poids lexico-négatif dans G', dans ce cas il n'existe pas de *retiming* répondant à la contrainte.

Preuve Si pour tout sommet u un plus court chemin p_u est trouvé, pour tout arc $e = (u, v) \in E$, il vérifie l'inégalité triangulaire : $\pi(v) \leq \pi(u) + d'(e)$ donc $d(e) + r(v) - r(u) \geq m(e)$.

Dans le cas contraire, l'algorithme de Bellman-Ford nous certifie que G' contient un circuit de poids lexico-négatif. Supposons qu'il existe un *retiming* légal r de G', cela voudrait dire que tous les circuits de G'_r ont un poids lexico-positif ou nul. Comme le *retiming* conserve le poids des circuits (voir le corollaire 1), nous en déduisons que tous les circuits de G' ont un poids lexico-positif ou nul, ce qui est une contradiction. Donc il n'existe pas de *retiming* légal de G', autrement dit il n'existe pas de *retiming* r tel que $\forall e = (u, v) \in E, d'_r(e) \geq_{lex} 0 \iff d'(e) + r(v) - r(u) \geq_{lex} 0 \iff d(e) - m(e) + r(v) - r(u) \geq_{lex} 0 \iff d_r(e) \geq_{lex} m(e)$. Il n'existe donc pas de *retiming* r de G qui respecte la contrainte de poids minimal m. □

Cette propriété sur les arcs liée au *retiming* légal est utilisée par Leiserson et Saxe afin de contraindre le poids d'un chemin : lorsque le poids d'un chemin p entre u et v doit être supérieur ou égal à k, il suffit d'ajouter un arc entre u et v de poids $d(p) - k$ et de trouver un *retiming* légal du graphe résultant (voir l'exemple de la section 1.2.3 pour de plus amples détails). La complexité de cet algorithme est dominée par la complexité de l'algorithme de Bellman-Ford, c'est-à-dire $O(|V||E|)$.

1.2.3 Un exemple d'algorithme de *retiming* : l'algorithme de Leiserson et Saxe

Dans [70], Leiserson et Saxe présentent la technique du *retiming*. Ils la conçoivent comme une transformation de graphe pour l'optimisation de circuits synchrones. Un circuit synchrone est un

circuit logique composé d'unités fonctionnelles, de registres et d'interconnections, régi par une horloge globale. Leiserson et Saxe modélisent un tel circuit par un graphe très proche de notre modèle non pipeliné : les sommets du graphe sont des unités fonctionnelles (portes logiques, additionneurs, etc.), les arcs sont des interconnections entre unités fonctionnelles, le poids sur les arcs est un nombre de registres séparant deux unités fonctionnelles, et le temps d'exécution est un délai de propagation à travers l'unité fonctionnelle. La figure 1.7 est un exemple de circuit synchrone tiré de l'article de Leiserson et Saxe. Sur cet exemple, le délai de propagation d'une unité fonctionnelle est indiqué à l'intérieur de celle-ci, et les registres sont représentés par les rectangles sur les arcs. Tout comme

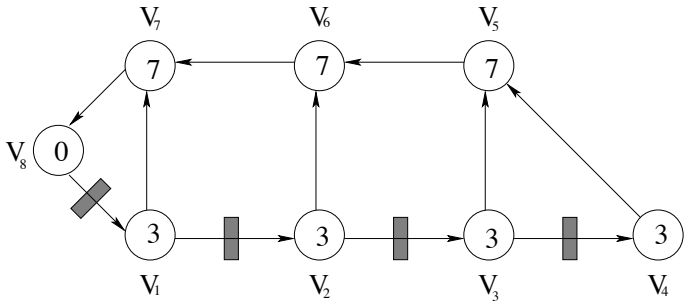


FIG. 1.7 – Un exemple de circuit synchrone tiré de [70].

dans notre modèle, tout graphe représentant un circuit synchrone n'a pas de circuit de poids nul (graphe correct), de plus il n'a pas d'arc de poids négatif (graphe légal) puisqu'un nombre de registres négatif n'a aucun sens. Dans un tel circuit, les registres jouent un rôle de tampon : à chaque « top » d'horloge ils stockent la valeur qui leur est fournie en entrée et envoient sur leur sortie la valeur stockée au « top » précédent. Ce rôle des registres a deux conséquences :

- le temps nécessaire à la stabilisation des données du circuit est égal au temps de propagation des données au travers de toutes les unités fonctionnelles séparant deux registres ;
- la présence de registres permet de pipeliner les calculs effectués par le circuit : à chaque « top » d'horloge il est possible d'introduire dans le circuit de nouvelles données qui traverseront « top » après « top » le circuit entier.

À partir de cette constatation, Leiserson et Saxe définissent la période d'horloge du circuit, qui est le temps de propagation maximal entre deux registres (ou, dans notre modèle, le temps d'exécution sur un chemin de poids nul). Cette période d'horloge est intéressante car elle est un indicateur de la fréquence maximale à laquelle il est possible de cadencer le circuit, donc de la performance de celui-ci : plus la période d'horloge est basse, plus le circuit pourra être cadencé rapidement. L'utilisation de *retiming* sur un circuit permet d'influer sur le nombre et la position des registres, donc sur la période d'horloge, tout en laissant le circuit fonctionnellement équivalent. L'objectif est alors de minimiser la période d'horloge afin de produire le circuit le plus performant possible. C'est ce que fait l'algorithme de Leiserson et Saxe : il permet de déterminer un *retiming* minimisant la période d'horloge d'un graphe.

L'exemple de la figure 1.7 a une période d'horloge de 24 (la longueur du chemin sans registres de V_4 à V_8). En utilisant l'algorithme de Leiserson et Saxe que nous allons présenter, nous obtenons le circuit de la figure 1.8 sur laquelle le *retiming* trouvé est indiqué. Ce nouveau circuit a une période d'horloge de seulement 13, c'est la période d'horloge optimale.

L'algorithme de Leiserson et Saxe est découpé en plusieurs parties construites de façon incrémentale. Nous présentons toutes ces parties telles qu'il est possible de les trouver dans [70] mais en utilisant nos notations. Une justification plus concise remplace certaines preuves, mais les preuves

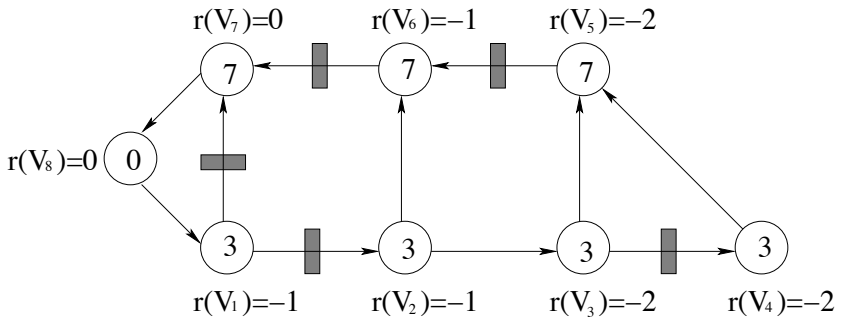


FIG. 1.8 – Le circuit de la figure 1.7 après minimisation de la période d'horloge.

complètes se trouvent dans l'article de Leiserson et Saxe. Nous commençons par l'algorithme CP qui calcule la période d'horloge d'un circuit.

Algorithme 3 (*CP*)

- soit $A(G)$ le sous-graphe des arcs de poids nul de G (arcs sans registres) ;
- par correction de G, $A(G)$ est acyclique, calculer une extension linéaire (ordre topologique) de $A(G)$;
- parcourir les sommets de $A(G)$ dans l'ordre topologique, pour chaque sommet v visité calculer la quantité suivante :
  - s'il n'y a pas d'arc entrant dans v, poser $\Delta(v) \leftarrow t_c(v)$;
  - sinon poser $\Delta(v) \leftarrow t_c(v) + \max\{\Delta(u) \mid e = (u, v) \in E \text{ et } d(e) = 0\}$;
- la période d'horloge $\Phi(G)$ est $\max_{v \in V} \Delta(v)$.

Cet algorithme calcule bien la période d'horloge, car pour chaque sommet $v \in V$, $\Delta(v)$ est égal au délai de propagation maximal d'un chemin de poids nul menant à v. Sa complexité est $O(|E|)$.

Nous définissons alors deux quantités qui vont nous permettre de caractériser la période d'horloge d'un circuit :

$$\begin{aligned} W(u, v) &= \min\{d(p) \mid p \text{ chemin de } u \text{ vers } v\} \\ D(u, v) &= \max\{t_c(p) \mid p \text{ chemin de } u \text{ vers } v \text{ et } d(p) = W(u, v)\} \end{aligned}$$

Ces deux quantités représentent respectivement le poids minimal d'un chemin reliant u à v, et le délai maximal de propagation sur un chemin de poids minimal reliant u à v. L'algorithme WD permet de calculer ces deux quantités pour toute paire de sommets.

Algorithme 4 (*WD*)

- pondérer chaque arc $e = (u, .) \in E$ par le vecteur $(d(e), -t_c(u))$;
- grâce à cette pondération, calculer le poids du plus court chemin joignant toute paire de sommets à l'aide d'un algorithme standard (Floyd-Warshall par exemple) en utilisant l'ordre lexicographique ;
- pour chaque poids (x, y) du plus court chemin entre deux sommets u et v, poser $W(u, v) \leftarrow x$ et $D(u, v) \leftarrow t_c(v) - y$.

Le plus intéressant est que ces quantités se comportent bien vis-à-vis du *retiming* :
- $W_r(u,v) = W(u,v) + r(v) - r(u)$, simplement parce que le lemme 1 nous dit que le poids de tous les chemins entre u et v varie de la même façon ;
- $D_r(u,v) = D(u,v)$, car le chemin de poids minimal ne change pas par *retiming* (tous les chemins entre u et v varient de la même quantité).

Il en résulte que la période d'horloge d'un circuit après tout *retiming* r est nécessairement égale à l'une des valeurs $D_r(u,v)$: la valeur maximale vérifiant $W_r(u,v) = 0$. L'idée pour produire un circuit ayant une période d'horloge inférieure ou égale à c est donc d'imposer $W_r(u,v) > 0$ lorsque $D_r(u,v) > c$, ceci est caractérisé par le théorème suivant (qui est le théorème 7 dans [70]) :

Théorème 1 (Faisabilité d'une période d'horloge) *Soit $G = (V, E, d)$ un circuit synchrone, c un entier positif quelconque, et r une fonction de V dans les entiers relatifs. Alors r est un* retiming *légal de G tel que $\Phi(G_r) \leq c$ si et seulement si :*

$$\begin{aligned} r(u) - r(v) &\leq d(e) && \text{pour tout arc } e = (u,v) \in E, \text{ et} \\ r(u) - r(v) &\leq W(u,v) - 1 && \text{pour tous sommets } u, v \in V \text{ tels que } D(u,v) > c \end{aligned}$$

La preuve est immédiate : la première condition est vérifiée par tout *retiming* légal, et la seconde exprime le fait que si $D(u,v) > c$ alors $W_r(u,v)$ doit être strictement positif, autrement la période d'horloge sera au moins égale à $D(u,v)$. De ce théorème nous tirons directement l'algorithme de minimisation :

Algorithme 5 (*OPT1*)

- calculer W et D en utilisant l'algorithme WD ;
- trier les éléments de D ;
- ajouter des arcs de poids $W(u,v) - 1$ entre toute paire de sommets (u,v) telle que $D(u,v) > c$;
- effectuer une recherche par dichotomie de la période d'horloge minimale parmi les éléments de D. Pour vérifier si une période d'horloge candidate c est atteignable par *retiming*, utiliser l'algorithme 2 de *retiming* sous contrainte pour trouver un *retiming* légal du graphe ;
- le *retiming* cherché est celui trouvé pour la période d'horloge minimale lors de la recherche dichotomique.

Dans cet algorithme, l'utilisation de l'algorithme 2 de *retiming* sous contrainte peut être remplacée par l'utilisation de l'algorithme FEAS de [70], spécifiquement conçu pour tester si une période d'horloge est atteignable par *retiming*, et dont la complexité est moindre. Dans tous les cas, nous obtenons le *retiming* produisant la période d'horloge minimale que nous notons $\Phi_{opt}(G)$.

Chapitre 2

Pipeline logiciel décomposé

2.1 Introduction

Allant de paire avec la montée en fréquence, chaque nouvelle version plus « puissante » d'un processeur propose généralement des capacités de plus en plus grandes d'exécution parallèle au niveau des instructions. Par exemple, les supercalculateurs Cray [19] offrent des unités vectorielles de calcul, le futur Itanium d'Intel [59] est doté de capacités VLIW (*Very Large Instruction Word*) et de support matériel pour l'exécution spéculative, et l'Emotion Engine de Toshiba/Sony [102], à mi-chemin entre les deux, possède deux coprocesseurs SIMD/VLIW. Ceci ne se limite pas aux processeurs spécialisés pour le calcul haute performance, puisque des produits grand public destinés à un usage plus général comme la série des Pentium d'Intel [58, 60] ou l'Athlon d'AMD [6] proposent plusieurs unités fonctionnelles, ainsi que plusieurs étages de pipeline. En d'autres termes, même les processeurs dits « séquentiels » ne sont plus une simple unité fonctionnelle exécutant les instructions à la suite les unes des autres en suivant un flot unique. Cependant, la disponibilité de ces ressources de calcul en quantité toujours plus importante pose un nouveau problème : les applications dédiées à ces machines doivent être à même d'exploiter suffisamment de parallélisme afin de fournir au processeur un flot soutenu d'instructions. Exploiter le parallélisme seulement à l'intérieur des blocs de base du programme n'est plus suffisant. Il est désormais nécessaire de considérer plusieurs blocs formant des structures plus complexes pouvant contenir des sauts ou des tests afin de découvrir une quantité de parallélisme toujours plus grande.

C'est donc naturellement que s'est développée une branche de la recherche consacrée à l'exploitation de parallélisme au niveau des instructions (ou entre instructions), aussi bien du point de vue matériel que logiciel (voir [55, Chap. 4] pour un état de l'art). L'exécution spéculative ou le support pour la prédication des instructions forment des solutions matérielles efficaces au problème posé par la présence de structures de contrôle complexes (voir par exemple les capacités de prédication de l'IA-64 [32]). D'un point de vue logiciel, il est possible, en tirant parti d'informations de plus haut niveau dans la chaîne de compilation, de déplacer les instructions à travers des structures de contrôle dont le comportement présente un caractère régulier ou facilement prévisible. C'est l'effet que produisent entre autres le déroulage de boucle ou l'ordonnancement par trace. C'est également le principe que le pipeline logiciel applique aux sauts d'une boucle : la boucle est restructurée afin de produire une nouvelle boucle dont chaque itération est constituée d'un ensemble d'instructions provenant de différentes itérations de la boucle d'origine.

Le problème du pipeline logiciel consiste à placer les instructions formant le corps de la boucle sur les ressources disponibles en respectant les dépendances, il ne considère pas le problème de l'allocation des registres qui n'intervient généralement qu'une fois les instructions placées. Ce problème

est NP-complet lorsque les ressources sont limitées (c'est-à-dire en nombre fini) même sans considérer l'allocation des registres. C'est pour cette raison qu'un grand nombre d'heuristiques ont été proposées suivant diverses stratégies pour s'attaquer au problème. Un état de l'art assez complet est proposé par Allan et al. dans [3]. Tous les algorithmes de pipeline logiciel ont néanmoins un objectif commun : ils recherchent une solution ayant la forme d'un ordonnancement cyclique, c'est-à-dire, à l'image de la boucle initiale, reproduisant le même calcul à chaque nouvelle itération. Allan et al. classifient ces algorithmes en trois principales catégories :

- le modulo scheduling ([67, 94]) et ses variantes ([93, 57, 76] par exemple) : le principe de cette approche est de s'autoriser des essais successifs jusqu'à aboutir a une solution. De façon pratique, l'algorithme commence par fixer un *intervalle de lancement* souhaité, c'est-à-dire une période pour l'ordonnancement cyclique final. Il essaie ensuite de résoudre le problème d'ordonnancement associé, si l'ordonnancement échoue, l'*intervalle de lancement* est augmenté et l'algorithme retourne à la résolution du problème d'ordonnancement associé. La convergence est assurée par le fait que pour un *intervalle de lancement* assez grand, on peut retrouver la solution séquentielle ;
- les algorithmes de reconnaissance de motif (voir [2, 92]) : l'idée suit une approche plus « instinctive » du pipeline logiciel, il s'agit ici d'ordonnancer au plus tôt les itérations successives de la boucle jusqu'à voir apparaître un motif cyclique dans l'ordonnancement (ce qui correspond à une sorte de déroulage de la boucle). Ici, aucune garantie n'est fournie quant au nombre d'itérations à ordonnancer avant la convergence, ni quant à la taille de l'ordonnancement final ;
- les algorithmes de décalage/ordonnancement (voir [61, 80, 13]) : ces algorithmes découpent le problème en deux parties, le déplacement des opérations à travers les différentes itérations d'une part, et l'ordonnancement à l'intérieur de la boucle d'autre part. Ils fonctionnent le plus souvent de manière itérative en déplaçant arbitrairement certaines opérations de la boucle puis en essayant de compacter le nouveau corps ainsi obtenu, jusqu'à ce qu'aucune amélioration ne soit plus possible. Là non plus aucune garantie n'est donnée quant à la vitesse de convergence.

Notre but ici est d'explorer plus en détails le concept du décalage/ordonnancement, en particulier nous voulons savoir plus précisément comment déplacer les instructions entre itérations peut aider à compacter le corps d'une boucle. En effet, comme l'explique B. Rau dans [93], avec l'approche classique du décalage/ordonnancement, « bien que ce mouvement des instructions puisse produire une amélioration de l'ordonnancement, savoir quelles opérations déplacer le long de l'arc de retour, dans quelle direction et combien de fois pour obtenir le meilleur résultat n'est pas toujours clair. [...] Dans quelle mesure, en pratique, cette méthode s'approche de l'optimal n'a toujours pas été étudié, et, en fait, même la notion d'optimal n'a pas été définie pour cette approche »[1]. Poursuivant les idées développées dans le cadre du pipeline logiciel décomposé (voir [48, 105, 10]), nous souhaitons montrer comment nous pouvons trouver directement en une étape de pré-calcul un « bon »[2] décalage afin d'améliorer au maximum la compaction de boucle. Les algorithmes de pipeline logiciel décomposé peuvent être considérés comme des algorithmes de décalage/ordonnancement particuliers n'utilisant que deux étapes seulement : une étape globale de déplacement des instructions suivie d'une étape d'ordonnancement. Le problème est découpé en un sous-problème d'ordonnancement cyclique sans contraintes de ressources (trouver le décalage), suivi d'un problème d'ordonnancement acyclique avec contraintes de ressources (compacter le corps de la boucle).

[1] *although such code motion can yield improvements in the schedule, it is not always clear which operations should be moved around the back edge, in which direction and how many times to get the best results. [...] How close it gets, in practice, to the optimal has not been studied, and, in fact, for this approach, even the notion of "optimal" has not been defined.*

[2] Nous employons ici les guillemets car la technique reste bien sûr une heuristique, et la notion de « bon » reste à définir. Même la compaction de boucle seule reste un problème NP-complet dans le cas de ressources limitées...

Le reste de ce chapitre est organisé de la manière suivante. Nous commençons, à la section 2.2, par reformuler les notions relatives au problème du pipeline logiciel, en particulier nous rappelons un certain nombre de résultats connus sur sa complexité, les bornes théoriques sur les performances, etc. Nous expliquons ensuite à la section 2.3 pourquoi la combinaison d'un décalage d'instructions avec une compaction de boucle donne de meilleurs résultats qu'une compaction seule. Le décalage permet de changer certaines dépendances non portées par la boucle en dépendances portées par la boucle et inversement. Ensuite, la compaction réordonnance le corps de la boucle en ne tenant compte que des dépendances indépendantes de la boucle, mais en considérant les détails de l'architecture cible. Une première optimisation consiste donc, à l'aide du décalage, à minimiser la longueur du chemin critique pour la compaction de boucle comme cela a été proposé dans [10] en utilisant un algorithme dû à Leiserson et Saxe (voir section 1.2.3). Nous souhaitons pousser cette idée plus loin en choisissant un décalage minimisant le nombre total de contraintes restant lors de la phase de compaction, c'est-à-dire minimisant le nombre total de dépendances indépendantes de la boucle. Cette idée sera l'objet de la section 2.4. La section 2.5 est dédiée à l'évaluation de notre technique comparée à la compaction seule, ainsi qu'à d'autres heuristiques classiques de pipeline logiciel décomposé sur des graphes de dépendance aléatoires et des modèles de machine complexes. Enfin, nous conclurons sur notre approche du pipeline logiciel décomposé à la section 2.6.

2.2 Le problème du pipeline logiciel

Comme précisé précédemment, notre but ici est de déterminer, dans le contexte des algorithmes de décalage/ordonnancement, comment déplacer les instructions de la meilleure manière afin de rendre la compaction aussi efficace que possible. Ainsi, nous devons être capables de juger de la qualité d'un ordonnancement, nous devons avoir une notion de qualité absolue d'une solution, une notion d'« optimal » ou de performance relative à un « optimum », ce qui nous permettra de choisir judicieusement notre décalage. Pour ceci, nous avons besoin d'un modèle aussi bien pour représenter une boucle que pour l'architecture cible de notre ordonnancement. Tous deux sont présentés en section 1.1.3, et dans le cas présent notre modèle se limite aux boucles simples. Nous supposons que les boucles sont sans branchements conditionnels. Dans le cas contraire, nous considérons que tout bloc formant une instruction conditionnelle forme une instruction atomique exécutée de façon inconditionnelle[3], nous nous limitons en outre au cas des dépendances uniformes.

D'un point de vue purement théorique, nous discutons de la performance de notre approche décomposée dans le cas d'un nombre fini d'unités fonctionnelles homogènes. En pratique cependant, notre algorithme peut être utilisé pour des modèles d'architectures plus sophistiqués (bien que la garantie théorique que nous donnons ne soit plus vraie). En effet, la technique de décalage d'instructions que nous développons n'est en fait que la première étape de l'algorithme et ne dépend pas du modèle d'architecture cible, mais seulement des contraintes de dépendance. Elle consiste juste à déplacer certaines instructions entre itérations de telle sorte que le chemin critique et le nombre de contraintes pour l'ordonnancement du corps soit minimisé. Les contraintes de ressources et donc les détails de l'architecture cible ne sont considérés que dans la deuxième étape, lorsque le corps de la boucle est compacté, et un ordonnanceur « agressif » spécifique à la machine considérée peut tout à fait être employé pour cette tâche. Pour l'évaluation expérimentale que nous avons menée (voir la section 2.5), nous avons justement considéré plusieurs types d'architectures, des machines à unités fonctionnelles simples, complexes, homogènes ou non, pipelinées ou non.

[3] Discuter de l'optimalité d'un ordonnancement pour une boucle contenant des instructions conditionnelles peut se révéler ardu, voir les travaux de Schwiegelshohn et al. [100].

2.2.1 Branchements conditionnels

Le problème des branchements conditionnels n'a pas été pleinement résolu à l'heure actuelle, bien que certains algorithmes de décalage/ordonnancement gèrent plus efficacement ce type de contraintes (voir par exemple [80, 105]). Mais nous devons garder à l'esprit que ces algorithmes fonctionnent par petites étapes permettant d'avoir un comportement conservatif vis-à-vis de ces branchements mais ne garantissent nullement la qualité du résultat. Notre approche radicalement différente ne peut pas profiter dans la même mesure d'une approche similaire. Le fait que l'ensemble d'une structure de branchement conditionnel puisse être éclaté sur plusieurs itérations par le décalage d'instructions ne permet pas une reconstruction efficace de celui-ci. Il ne reste donc comme recours que la technique de la *if-conversion* (voir [106]), utilisée également par le *modulo scheduling*. Cette technique considère le résultat du test comme un prédicat et éclate la structure conditionnelle, en rendant l'exécution de chaque instruction des deux branches dépendante du résultat du prédicat. Ceci s'avère particulièrement adapté aux machines offrant un support matériel pour la prédication (voir [96, 32]), et il semble qu'elle trouve ses défenseurs même en l'absence d'un tel support (voir [107]). Néanmoins, ne sachant pas à l'avance quelle branche sera choisie, il reste difficile de proposer une transformation la plus proche de l'optimal possible. De plus, dans le cadre de notre approche et de façon générale, afin de profiter au mieux de l'exclusion mutuelle des deux branches, l'ordonnanceur sous-jacent doit prendre en compte le partage possible des ressources entre deux opérations appartenant à deux branches distinctes. Finalement, nous restons sur ce point à égalité avec le *modulo scheduling* : aucune des deux techniques n'apporte de réponse parfaite au problème des branchements conditionnels.

2.2.2 Formulation du problème

Nous considérons le problème de l'ordonnancement d'une boucle pouvant comporter un nombre très important d'itérations sur un processeur offrant certaines capacités limitées de parallélisme. En d'autres, termes nous sommes dans un contexte de pipeline logiciel, c'est-à-dire qu'aussi bien une exécution en parallèle de toutes les itérations de la boucle qu'un déroulage complet de celle-ci sont exclus. Notre boucle est représentée par un graphe de dépendance simple (mono-dimensionnel) pondéré sur ses sommets par t_r et t_c, respectivement le temps de réservation d'une ressource et le temps d'exécution pour l'opération associée, et sur ses arcs par d, la distance de dépendance entre deux opérations, et l, la latence minimale entre deux opérations dépendantes (nombre de cycles sur l'architecture cible). Ce modèle est présenté plus en détails en section 1.1.3.

Au niveau de la boucle, un arc e correspond à une dépendance indépendante de la boucle si $d(e) = 0$, et portée par la boucle sinon. Une dépendance indépendante de la boucle est toujours dirigée d'une instruction u vers une instruction v qui la suit textuellement. De plus, un graphe correspondant à une boucle ne possède pas de circuit de poids négatif ou nul (graphe correct).

Notre but est de déterminer un ordonnancement pour toutes les opérations (v, k), c'est-à-dire une fonction $\sigma : V \times \mathbb{Z} \to \mathbb{Z}$ respectant les contraintes de dépendance :

$$\forall e = (u, v) \in E,\ \forall k \geq 0, \quad \sigma(u, k) + l(e) \leq \sigma(v, k + d(e)) \tag{2.1}$$

et les contraintes de ressources de notre problème : par exemple, si p unités fonctionnelles homogènes non pipelinées sont disponibles, pas plus de p opérations ne doivent être exécutées à un instant donné. Des schémas de ressources plus complexes peuvent être représentés par l'intermédiaire de tables de réservation (voir par exemple [93] pour une description détaillée), cependant rappelons que nos garanties théoriques de performance ne tiennent que pour un modèle à ressources homogènes.

La performance d'un ordonnancement est mesurée par sa période moyenne λ définie par :

$$\lambda = \liminf_{N\to\infty} \frac{\max\{\sigma(v,k) + t_c(v) \mid v \in V,\ 0 \leq k < N\}}{N}$$

qui est le temps moyen nécessaire à l'exécution d'une itération de la boucle. Parmi tous les ordonnancements, ceux exhibant un motif cyclique sont particulièrement intéressants puisque l'on peut les décrire sous la forme d'une boucle. Un ordonnancement cyclique σ est un ordonnancement tel que $\sigma(v,k) = a_v + \lambda k$, avec $a_v \in \mathbb{Z}$ et $\lambda \in \mathbb{N}$. L'ordonnancement σ a une période de λ : le même schéma d'exécution est répété toutes les λ unités de temps. A l'intérieur de chaque période, une et une seule instance de chaque opération est démarrée : λ est pour cette raison également appelé *intervalle de lancement* (*initiation interval* dans la littérature). Dans le cas d'un ordonnancement cyclique de période positive, λ est égal à la période moyenne de l'ordonnancement.

2.2.3 Bornes inférieures à la période moyenne

La période moyenne de tout ordonnancement (cyclique ou non) est limitée à la fois par les contraintes de ressources et par les contraintes de dépendance. Nous notons λ_∞ (resp. λ_p) la période moyenne minimale atteignable par un ordonnancement (cyclique ou non) avec une infinité de ressources (resp. p ressources homogènes). Bien sûr, $\lambda_\infty \leq \lambda_p$. Nous avons les bornes suivantes sur λ_∞ et λ_p :

$$\lambda_p \geq \lambda_\infty \geq \max\{\frac{l(c)}{d(c)} \mid c \text{ circuit de } G\} \text{ (contraintes de dépendances)} \quad (2.2)$$

$$\lambda_p \geq \frac{\sum_{v\in V} t_r(v)}{p} \text{ (contraintes de ressources)} \quad (2.3)$$

La preuve de ces deux bornes se trouve dans [25]. Remarquons, que la seconde contrainte n'est pertinente que dans le cas de ressources homogènes, dans le cas de modèles plus complexes il devient nécessaire de la raffiner – par exemple en prenant le maximum d'un calcul similaire sur chaque type de ressource. Ces bornes sont connues dans la littérature sous les noms respectifs de RecMII (pour *Recurrence Minimum Initiation Interval*) et ResMII (pour *Resource Minimum Initiation Interval*).

Sans contraintes de ressources, le problème d'ordonnancement peut être résolu en temps polynomial (voir [25] pour un algorithme détaillé). En effet, λ_∞ est alors égal à RecMII et il existe un ordonnancement cyclique optimal (éventuellement avec un *intervalle de lancement* fractionnaire ou un déroulage de la boucle si λ_∞ n'est pas entier). Un tel ordonnancement peut être déterminé par un algorithme standard de ratio minimal (comme celui de [38, pp. 636-641]).

À l'inverse, lorsque des contraintes de ressources s'ajoutent au problème, déterminer un ordonnancement ayant une période moyenne minimale est NP-difficile (son appartenance ou non à NP est encore un problème ouvert). Lorsque les solutions sont restreintes aux ordonnancements cycliques, le problème devient NP-complet. Le lecteur intéressé pourra se reporter à [25, 54] pour un état de l'art du problème de l'ordonnancement cyclique.

2.3 Décalage d'instructions et compaction de boucle

Dans cette section, nous expliquons comment le décalage d'instructions peut être utilisé afin d'améliorer la performance des techniques de compaction de boucle. Nous formalisons dans un premier temps le problème de la compaction de boucle, et rappelons quelques résultats sur ses performances. Notre but ici est d'expliquer, de façon théorique aussi bien qu'intuitive, pourquoi

transformer le graphe en réduisant son chemin critique en vue de la compaction peut réduire l'*intervalle de lancement* de la boucle.

2.3.1 Performance de la compaction seule

La compaction de boucle consiste à ordonnancer le corps de la boucle sans tenter de recouvrir ou de mélanger les itérations successives entre elles. La compaction s'applique à un graphe de dépendance légal (sans arc de poids négatif). Rappelons qu'un tel graphe peut être obtenu par *retiming* à partir de tout graphe correct. De plus, cette contrainte de légalité est d'autant moins gênante que tout graphe obtenu par analyse d'un code écrit dans un langage séquentiel classique est légal (du moins pour une seule boucle). Le mécanisme de la compaction distingue les dépendances indépendantes de la boucle des autres, nous définissons donc un sous-graphe particulier du graphe de dépendance regroupant les contraintes pour la compaction.

Définition 19 (sous-graphe des arcs de poids nul) *Soit $G = (V, E, d)$ un graphe de dépendance, nous notons $A(G) = (V, E', d)$ le sous graphe de G défini par $E' = \{e \in E \mid d(e) = 0\}$ (sous-graphe des arcs de poids nul de G, ou encore sous-graphe induit par les dépendances indépendantes de la boucle).*

Le graphe $A(G)$ est acyclique puisque G n'a pas de circuit de poids nul (en fait, G n'a pas de circuit de poids négatif ou nul, mais nous n'avons pas besoin d'autant d'information). $A(G)$ peut donc être ordonnancé en utilisant les techniques d'ordonnancement classiques pour les graphes acycliques dirigés (par exemple l'ordonnancement par liste). Le nouveau motif créé pour le corps de la boucle, c'est-à-dire le résultat de l'ordonnancement acyclique, est alors répété par les itérations successives de la boucle, formant ainsi notre ordonnancement cyclique. À l'intérieur du corps de la boucle, les contraintes de ressources et de dépendances sont respectées par l'ordonnanceur acyclique, tandis que la séparation des motifs successifs (les motifs des itérations successives ne se recouvrent pas) assure le respect de ces contraintes entre deux itérations distinctes.

Le seul point sur lequel nous devons être prudents est que nous devons choisir un *intervalle de lancement* suffisamment grand pour ne générer aucun conflit entre deux motifs successifs. Une hypothèse conservative est de laisser après la date de début de toute opération u un temps fixe, suffisant pour respecter toutes les contraintes de ressources et de dépendance dans lesquelles elle est impliquée. Pour cela, nous pouvons choisir $t_\Delta(u) = max(\{t_r(u)\} \cup \{l(e) | e = (u, .) \in E\})$ et nous montrons dans la preuve de l'algorithme de compaction que cette grandeur est suffisante. Pourtant, cette grandeur est une sur-approximation grossière du temps nécessaire à la fin de chaque opération. Une meilleure solution serait, une fois l'ordonnancement réalisé, de choisir l'*intervalle de lancement* le plus petit vérifiant les contraintes de dépendance et de ressources (c'est ce que nous faisons en section 2.5, lors de la réalisation pratique de l'algorithme). Cependant, dans le cadre de notre discussion théorique, nous avons besoin d'exprimer des bornes sur l'*intervalle de lancement* ne dépendant pas des dates produites par l'ordonnancement acyclique, c'est pourquoi nous tenons à conserver t_Δ.

Algorithme 6 (*Compaction de boucle*)

1. construire $A(G) = (V, E')$ où $E' = \{e \in E \mid d(e) = 0\}$;
2. construire un ordonnancement σ_a de $A(G)$, en tenant compte des contraintes de ressources (on pourra utiliser par exemple un ordonnancement par liste) ;
3. calculer la durée d'exécution (*makespan*) de σ_a : $\lambda_\sigma = \max_{u \in V}(\sigma_a(u) + t_\Delta(u)) - \min_{v \in V} \sigma_a(v)$;
4. l'ordonnancement cyclique cherché est alors : $\forall u \in V,\ \forall k \in \mathbb{N},\ \sigma(u, k) = \sigma_a(u) + \lambda_\sigma k$.

Preuve La correction de cet algorithme est intuitivement évidente : on ordonnance avec un algorithme correct le corps, puis on met les uns à la suite des autres les motifs ainsi créés. La preuve plus formelle repose, entre autres, sur l'hypothèse de légalité du graphe de dépendance. Nous supposons que la construction de l'ordonnancement se fait par un algorithme correct, c'est-à-dire respectant les contraintes de ressources, et les contraintes de dépendance exprimées par $A(G)$. Il reste alors à montrer que les autres contraintes sont satisfaites. Nous supposons pour cela que le graphe de dépendance est légal, donc que toute dépendance non présente dans $A(G)$ a un poids strictement positif. Nous avons $\lambda_\sigma \geq 0$ car $\forall u \in V, t_\Delta(u) \geq t_r(u) \geq 0$ par définition de $t_\Delta(u)$. Et pour tout arc $e = (u, v) \in E \backslash E'$, nous avons également :

$$\begin{aligned}
\sigma(v, k + d(e)) &= \sigma_a(v) + \lambda_\sigma(k + d(e)) \\
&= \sigma_a(v) + \lambda_\sigma k + \left(\max_{u \in V}(\sigma_a(u) + t_\Delta(u)) - \min_{v \in V} \sigma_a(v)\right) d(e) \\
&\geq \sigma_a(v) - \min_{v \in V} \sigma_a(v) + \lambda_\sigma k + \sigma_a(u) + t_\Delta(u) \\
&\geq \lambda_\sigma k + \sigma_a(u) + t_\Delta(u) \\
&\geq \sigma(u, k) + t_\Delta(u)
\end{aligned}$$

Les contraintes de dépendance sont donc bien respectées entre les motifs successifs. Enfin, il faut également montrer qu'il n'y a pas de conflit de ressources entre deux motifs successifs. Soient u et v deux sommets de G, ainsi que i et j tels que i<j, nous avons :

$$\begin{aligned}
\sigma(v, j) - \sigma(u, i) &= \sigma_a(v) + \lambda_\sigma j - \sigma_a(u) + \lambda_\sigma i \\
&= \sigma_a(v) - \sigma_a(u) + \left(\max_{u \in V}(\sigma_a(u) + t_\Delta(u)) - \min_{v \in V} \sigma_a(v)\right)(j - i) \\
&\geq \sigma_a(u) - \sigma_a(u) + t_\Delta(u) + \sigma_a(v) - \min_{v \in V} \sigma_a(v) \\
&\geq t_\Delta(u)
\end{aligned}$$

Donc toute ressource utilisée par une opération d'un motif donné est libérée avant le début de toute opération d'un motif suivant. Les contraintes de ressources entre motifs successifs sont donc également vérifiées. □

Qu'est-il possible d'espérer d'une telle technique ? A cause des contraintes de dépendance, la compaction de boucle est limitée par le chemin critique de $A(G)$, c'est-à-dire les chemins p de durée maximale.

Définition 20 (Chemin critique pour la compaction) *Soit P l'ensemble des chemins du graphe $A(G)$. Nous définissons $\Phi(G)$ comme la « durée » maximale d'un chemin de P :*

$$\Phi(G) = \max_{p=(u_1,\ldots,u_n) \in P} \left(\left(\sum_{i=1}^{n-1} l(u_i, u_{i+1})\right) + t_\Delta(u_n)\right) = \max_{p=(u_1,\ldots,u_n) \in P} (l(p) + t_\Delta(u_n))$$

Nous appelons chemin critique de $A(G)$ tout chemin de durée $\Phi(G)$.

Pour cette définition, nous utilisons la même notation que pour la période d'horloge d'un graphe (voir section 1.2.3). En effet, dans notre modèle non pipeliné (dans lequel s'applique l'algorithme de Leiserson et Saxe), $l(u_i, u_{i+1}) = t_c(u_i)$ et $t_\Delta(u_n) = t_c(u_n)$, la « durée » du chemin critique de G est donc égale à la période d'horloge du circuit associé à G. Cette définition est donc une généralisation de la définition de la période d'horloge et correspond à la même notion. En outre,

elle correspond bien à l'*intervalle de lancement* minimal que pourra trouver la compaction : tout d'abord l'ordonnancement acyclique doit respecter les contraintes exprimées par $A(G)$ et toute paire d'opérations (u_i, u_{i+1}) reliées par un arc dans ce graphe devra devra être séparée d'au moins $l(u_i, u_{i+1})$ cycles, ensuite la compaction laisse encore t_Δ cycles pour les dernières opérations (les puits de $A(G)$), afin de ne pas générer de conflits avec le motif suivant. Ceci nous donne donc une borne inférieure à la performance de la compaction de boucle : $\lambda_\sigma \geq \Phi(G)$. Pour la borne supérieure, nous séparons l'étude des modèles pipeliné et non pipeliné, qui donnent des résultat différents (mais du même ordre).

Modèle non pipeliné

Les résultat dans le modèle non pipeliné sont tirés de [10]. L'avantage de ce modèle est qu'il colle parfaitement aux modèles classiques pour l'ordonnancement par liste, ainsi tous les résultats existants peuvent être appliqués tels quels. Dans le modèle non pipeliné, nous avons $\forall u \in V, t_r(u) = t_c(u)$, et $\forall e = (u, .) \in E, l(e) = t_c(u)$.

Lemme 2 (Performance de la compaction (cas non pipeliné)) *Soit un graphe de dépendance G et un ordonnancement cyclique σ sur p ressources obtenu par compaction de G en utilisant un ordonnancement par liste, alors :*

$$\begin{aligned} \lambda_\sigma &\leq \frac{1}{p}\left(\sum_{u\in V} t_r(u) + (p-1)\Phi(G)\right) \\ &\leq \lambda_p + \Phi(G) - \frac{1}{p}\Phi(G) \end{aligned} \tag{2.4}$$

Preuve La preuve peut être trouvée dans [25], elle se fait en utilisant la technique de Graham pour la performance de l'ordonnancement par liste. Nous la reformulons ici.

Considérons notre ordonnancement cyclique, il est composé de motifs périodiques de durée λ_σ obtenus grâce à un ordonnancement par liste de $A(G)$. À chaque instant durant l'exécution d'un motif, un certain nombre de ressources sont disponibles et le reste est réservé par l'exécution d'une opération. De plus, la somme des ressources disponibles et occupées à cet instant fait un total de p (l'ensemble des ressources). En faisant la somme sur l'ensemble du motif, nous obtenons un total de disponibilité et d'occupation. La performance de notre ordonnancement est donc :

$$\lambda_\sigma = (\text{Disponibilité} + \text{Occupation})/p$$

L'exécution d'une instruction u de G occupe une ressource pendant $t_r(u)$ cycles, toutes les instructions de G appartiennent également à $A(G)$ et chaque motif contient exactement une instance de chacune d'entre elles. Le total d'occupation est donc :

$$\text{Occupation} = \sum_{u\in V} t_r(u)$$

Il nous reste a prouver que le total de disponibilité n'est pas arbitrairement grand. Pour cela, considérons une opération u_n terminant le motif, c'est-à-dire telle que $\sigma_a(u_n)+t_\Delta(u_n) = \max_{u\in V}(\sigma_a(u)+t_\Delta(u))$. Cette opération occupe une ressource durant les $t_\Delta(u_n)$ derniers instants du motif (car dans le modèle non pipeliné, $t_r(u_n) = t_\Delta(u_n)$), donc durant chacun de ces instants, au plus $(p-1)$ ressources sont disponibles. Regardons alors ce qu'il en est aux instants précédents. Considérons le plus grand instant t inférieur à $\sigma_a(u_n)$ durant lequel au moins une ressource est disponible. À cet

instant, soit un des ancêtres de u_n dans $A(G)$ est en cours d'exécution, soit u_n est prête. Si u_n est prête, alors elle aurait du être ordonnancée à l'instant t, la règle de tout ordonnancement par liste étant d'ordonnancer les opérations prêtes dès qu'une ressource est disponible. Donc, un ancêtre u_i de u_n est en cours d'exécution (autrement dit, il existe $u_i \in V$ et $e = (u_i, .) \in E$ sur un chemin de u_i à u_n tel que $\sigma_a(u_i) \leq t < \sigma_a(u_i) + l(e)$). De plus, de $\sigma_a(u_i)$ à t, au plus $(p-1)$ ressources sont disponibles (car dans le modèle non pipeliné, $t_r(u_i) = l(e)$), et entre t et $\sigma_a(u_n) - 1$ toutes les ressources sont occupées (par définition de t). En poursuivant le même raisonnement sur u_i et ses ancêtres, nous remontons jusqu'au début du motif et nous obtenons un chemin $(u_1, \ldots, u_i, \ldots, u_n)$ de $A(G)$ tel que le total de disponibilité vérifie :

$$\text{Disponibilité} \leq (p-1)\left(\sum_{i=1}^{n-1} l(u_i, u_{i+1}) + t_\Delta(u_n)\right) \leq (p-1)\Phi(G)$$

La figure 2.1 illustre sur un exemple de motif l'identification d'un chemin « couvrant » toutes les

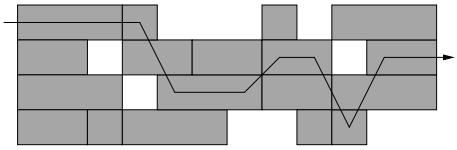


FIG. 2.1 – Illustration de la technique de Graham.

ressources disponibles. De l'équation précédente, nous déduisons que :

$$\lambda_\sigma \leq \frac{1}{p}\left(\sum_{u \in V} t_r(u) + (p-1)\Phi(G)\right)$$

□

Modèle pipeliné

Dans le cas pipeliné, il s'avère que la méthode de Graham s'applique toujours. Néanmoins, le modèle n'étant plus tout à fait le même, la borne obtenue sera différente.

Lemme 3 (Performance de la compaction (cas pipeliné)) *Soit un graphe de dépendance G, et un ordonnancement cyclique σ sur p ressources obtenu par compaction de G en utilisant un ordonnancement par liste, alors :*

$$\begin{aligned} \lambda_\sigma &\leq \frac{1}{p}\left(\sum_{u \in V} t_r(u) + \left(p - \frac{1}{\max_{u \in V} t_\Delta(u)}\right)\Phi(G)\right) \\ &\leq \lambda_p + \Phi(G) - \frac{1}{p\,(\max_{u \in V} t_\Delta(u))}\Phi(G) \end{aligned} \tag{2.5}$$

Preuve La preuve est quasiment identique à celle du lemme 2. Dans le modèle pipeliné, les équation suivantes tiennent toujours avec les mêmes justifications :

$$\begin{aligned} \lambda_\sigma &= (\text{Disponibilité} + \text{Occupation})/p \\ \text{Occupation} &= \sum_{u \in V} t_r(u) \end{aligned}$$

En revanche, pour le total de disponibilité nous n'avons plus égalité du temps de réservation avec la latence des arcs sortants. Autrement dit, la ressource requise pour l'exécution d'une opération peut très bien être libérée avant que le temps de latence ne se soit écoulé. La seule hypothèse raisonnable que nous pouvons faire est qu'une opération utilise une ressource pendant au moins un cycle lors de son lancement. Il en résulte que, pour chaque opération présente sur le chemin « couvrant » les ressources disponibles, il y a au plus $(p-1)$ ressources disponibles à son lancement et p durant le reste de son exécution. La figure 2.2 illustre sur le même motif qu'en figure 2.1 l'identification d'un

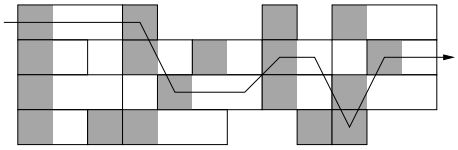


FIG. 2.2 – Illustration de la technique de Graham (modèle pipeliné).

chemin « couvrant » toutes les ressources disponibles. Sur cette figure, le temps de réservation est toujours égal à 1, donc beaucoup plus de ressources sont disponibles sur le chemin critique. Nous déduisons de tout ceci que sur un chemin de « durée » L au moins $L/\max_{u\in V} t_\Delta(u)$ opérations sont démarrées, et nous obtenons une borne sur le total de disponibilité :

$$\begin{aligned}\text{Disponibilité} &\leq p\left(\sum_{i=1}^{n-1} l(u_i,u_{i+1}) + t_\Delta(u_n)\right) - \frac{\sum_{i=1}^{n-1} l(u_i,u_{i+1}) + t_\Delta(u_n)}{\max_{u\in V} t_\Delta(u)} \\ &\leq \left(p - \frac{1}{\max_{u\in V} t_\Delta(u)}\right)\Phi(G)\end{aligned}$$

La fin est identique à la preuve du lemme 2. □

Bien sûr, la borne est maintenant plus grossière que la borne du cas non pipeliné. Cependant nous devons garder en tête que dans le cas pipeliné $t_r(u)$ et $l(e)$ sont inférieurs à $t_c(u)$ (justement parce que le processeur est pipeliné), il ne faut donc pas comparer ces deux bornes symboliquement puisque les grandeurs qu'elles mettent en jeu sont différentes. En particulier, aussi bien le temps d'occupation des ressources $\left(\sum_{u\in V} t_r(u)\right)$ que la « durée » du chemin critique sont raccourcis par le pipeline.

Remarque sur la performance de la compaction

Aussi bien dans le cas non pipeliné que pipeliné, la performance de la compaction seule est limitée par $\Phi(G)$, la « durée » du chemin critique de $A(G)$. Pour le sous-problème de l'ordonnancement acyclique, $\Phi(G)$ est une borne inférieure à la performance et, bien que l'ordonnancement par liste soit une heuristique garantie à un ratio inférieur à 2, ici malheureusement $\Phi(G)$ n'a – *a priori* – rien à voir avec l'*intervalle de lancement* minimum λ_p (en effet, cet intervalle dépend des circuits de dépendance et non du chemin critique de $A(G)$). C'est pour cette raison que la compaction de boucle seule peut être arbitrairement mauvaise.

Exemple Nous illustrons les différentes techniques utilisées tout au long de ce chapitre sur le même exemple donnée en figure 2.3. Afin de discuter du processus d'ordonnancement, nous avons choisi un modèle d'architecture très simple similaire au processeur LANai 3.0 développé par Myricom [81] : une simple unité fonctionnelle pipelinée à quatre étages pour laquelle les opérations ont un temps de

réservation d'un cycle et une latence d'un ou deux cycles selon le type d'opération. Dans cet exemple, la latence est la même sur tous les arcs sortant d'un sommet, et est égale au temps de calcul de ce sommet. Afin de simplifier la figure, nous avons donc représenté la latence sur les sommets du graphe. Le graphe acyclique $A(G)$ considéré pour la compaction de boucle est donné en figure 2.4. Sur cet exemple la compaction n'aide pas énormément, presque toutes les opérations doivent être

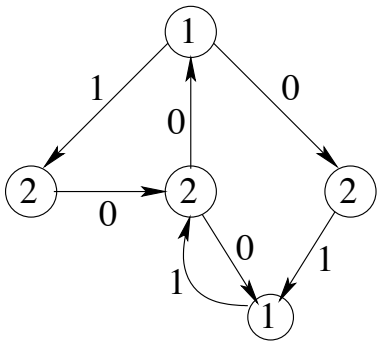


FIG. 2.3 – Graphe à ordonnancer.

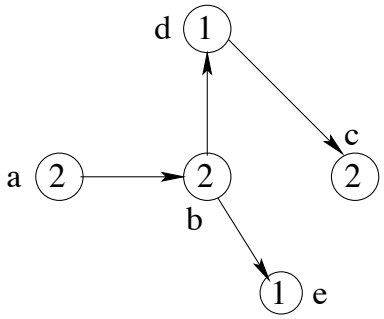


FIG. 2.4 – Graphe acyclique à compacter.

ordonnancées de façon séquentielle : d'abord a, puis b deux cycles plus tard, puis d encore deux cycles plus tard, et enfin c un cycle après. Un seul des cycles inoccupés dus à la latence peut être rempli par une opération en ordonnançant e juste après c. L'*intervalle de lancement* résultant est de 7 cycles.

Remarque : comme nous l'avons déjà remarqué lors de la présentation de l'algorithme 6 de compaction, construire un ordonnancement cyclique dont l'*intervalle de lancement* est plus petit que la longueur de l'ordonnancement acyclique calculé par l'algorithme est possible si l'on autorise les motifs d'itérations successives à se recouvrir. En effet, au lieu d'attendre la fin d'un motif avant de démarrer le suivant (en choisissant la longueur de l'ordonnancement acyclique comme *intervalle de lancement*), nous pouvons choisir le plus petit *intervalle de lancement* ne violant aucune contrainte. Cette optimisation est appelée compaction de boucle avec recouvrement (voir également [56]). Nous reviendrons à cette technique lors de nos expérimentations. Cependant, dans le cas général il n'est pas toujours possible d'améliorer l'*intervalle de lancement* par recouvrement, et la discussion précédente tient toujours. Revenons à notre exemple afin d'illustrer cette notion.

Exemple (suite) Il était possible de trouver une autre solution, ayant également 7 cycles d'*intervalle de lancement* en utilisant le recouvrement : si nous inversons c et e dans l'ordonnancement précédent, nous obtenons un ordonnancement acyclique dont la longueur est 8 (à cause de la latence de c, nous devons laisser un cycle vide à la fin). Pourtant, la dépendance est satisfaite car e (qui dépend de c) n'est pas ordonnancé en début d'itération, une nouvelle itération peut donc être démarrée tous les 7 cycles (voir la figure 2.7 (a)).

Notre but est de contourner la limite de la compaction due au chemin critique en entremêlant les itérations successives de la boucle (grâce au *retiming*), de telle sorte que le graphe résultant $A(G)$ – le sous-graphe des dépendances indépendantes de la boucle – soit optimisé pour la compaction.

2.3.2 Décalage d'instructions (*retiming*)

Nous montrons dans cette section les liens très forts entre *retiming* et pipeline logiciel. En particulier, nous précisons pourquoi l'utilisation de *retiming* peut permettre l'amélioration des performances de la compaction de boucle. Comme nous l'avons vu à la section 1.1.3, un *retiming* est un déplacement des opérations de la boucle dans leur domaine d'itération. Appliquer un *retiming* r à un graphe de dépendance $G = (V, E, d)$ revient à retarder l'exécution de toute opération (u, i) de $r(u)$ itérations. Ceci transforme le graphe G en un graphe $G_r = (V, E, d_r)$ tel que $\forall e = (u, v) \in E, d_r(u) = d(u) + r(v) - r(u)$. Et puisque le *retiming* retarde toute opération v de $r(v)$ itération, un ordonnancement cyclique associé à un graphe après *retiming* sera de la forme $\sigma(v, k) = a_v + \lambda(k + r(v))$. Rappelons qu'un *retiming* ne change pas le poids des circuits d'un graphe.

Lien entre *retiming*, compaction et ordonnancement cyclique Nous remarquons ici que tout ordonnancement cyclique (obtenu par exemple par modulo scheduling, mais de façon plus générale par toute technique de pipeline logiciel) correspond à la combinaison d'un *retiming* particulier r du graphe de dépendance G et d'un ordonnancement acyclique de $A(G_r)$. Considérons un ordonnancement cyclique $\sigma(u, k) = \lambda k + a_v$, en écrivant le résultat de la division euclidienne de a_v par λ, nous obtenons $a_v = \lambda r(v) + a'_v$ avec $0 \leq a'_v < \lambda$. Les valeurs $r(v)$ et a'_v sont respectivement appelés numéro de colonne et de ligne dans la terminologie du pipeline logiciel décomposé (voir [105]). Puisque les contraintes de dépendances sont satisfaites (sinon l'ordonnancement n'est pas valide), pour tout arc $e = (u, v) \in E$ et k positif nous avons :

$$\begin{array}{rrcl} & \sigma(v, k + d(e)) & \geq & \sigma(u, k) + l(e) \\ \iff & \lambda(k + d(e)) + a_v & \geq & \lambda k + a_u + l(e) \\ \iff & \lambda k + \lambda d(e) + \lambda r(v) + a'_v & \geq & \lambda k + \lambda r(u) + a'_u + l(e) \\ \iff & \lambda(d(e) + r(v) - r(u)) & \geq & l(e) + a'_u - a'_v \end{array} \quad (2.6)$$

Ce qui signifie que $d(e) + r(v) - r(u) = 0 \Rightarrow a'_v \geq a'_u + l(e)$, donc a' est bien un ordonnancement acyclique de $A(G_r)$ puisqu'il satisfait les contraintes de précédence exprimées par les arcs de poids nul de G_r. La seule différence entre cet ordonnancement cyclique et un ordonnancement cyclique obtenu par compaction est la contrainte sur λ : pour tout sommet v du graphe, $0 \leq a'_v < \lambda$, en d'autres termes, λ peut être inférieur à $\max_{u \in V}(a'_u + t_\Delta(u)) - \min_{v \in V} a'_v$. Cet ordonnancement acyclique correspond en fait à un motif obtenu par compaction dont on autorise le recouvrement avec les motifs suivants tant que les contraintes de ressource et de dépendance ne sont pas violées. Notons toutefois que λ reste toujours supérieur à toutes les dates, sinon cela signifie que l'ordonnancement correspond à un *retiming* différent.

Revenons alors à l'inégalité 2.6, si nous nous restreignons à des latences toutes positives, donc $l(e) \geq 0$, sachant que $0 \leq a'_v < \lambda$ nous en déduisons que :

$$\begin{array}{rrcl} & \lambda(d(e) + r(v) - r(u)) & > & -\lambda \\ \iff & d(e) + r(v) - r(u) & \geq & 0 \end{array}$$

puisque $d(e) + r(v) - r(u)$ est une valeur entière. Le *retiming* déterminé par la division euclidienne est donc légal.

En résumé, tout ordonnancement cyclique dans un modèle à latences positives correspond à la composition d'un *retiming* légal, d'une compaction de boucle et d'un recouvrement des motifs successifs dans la limite des dates de départ des opérations.

Remarque sur les latences négatives Revenons à l'équation 2.6, et autorisons cette fois les latences négatives :

$$\begin{aligned} & \lambda(d(e) + r(v) - r(u)) \geq l(e) + a'_u - a'_v \\ \iff \quad & \lambda(d(e) + r(v) - r(u)) > l(e) - \lambda \\ \iff \quad & d(e) + r(v) - r(u) > \frac{l(e)}{\lambda} - 1 \end{aligned} \tag{2.7}$$

Rien n'empêche alors le *retiming* d'être illégal, mais dans ce cas, à cause de l'inégalité stricte, la latence doit être négative. Le problème avec un *retiming* illégal est qu'il peut correspondre à une solution très particulière dans laquelle λ est borné, en effet en reprenant l'équation précédente avec $d(e) + r(v) - r(u) < 0$ nous obtenons :

$$\lambda \leq \frac{l(e) + a'_u - a'_v}{d(e) + r(v) - r(u)} \tag{2.8}$$

En étudiant la latence et le poids dans cette configuration, nous pouvons comprendre comment ceci est possible. Lorsque $d(e) + r(v) - r(u) < 0$, cela signifie que $d(e) + r(v) - r(u) \leq -1$ car le poids après *retiming* est entier. Autrement dit, une opération u dépend d'une opération v exécutée dans une itération future. Cette configuration apparemment dénuée de sens est justifiée par la latence négative : l'opération dépendante est exécutée à la fin d'une itération et son exécution se prolonge dans l'itération suivante jusqu'à ce que l'opération source ait résolu la dépendance. Dans cette situation, il n'est plus toujours possible de fixer λ une fois l'ordonnancement acyclique effectué, si les opérations concernées par la dépendance de poids négatif ont été mal placées alors la borne sur λ peut très bien être inférieure à $\max_{u \in V}(a'_u + t_\Delta(u)) - \min_{v \in V} a'_v$. Autrement dit, la compaction peut échouer et ne trouver aucune solution. C'est pour cette raison que dès le départ nous avons toujours imposé à notre graphe de dépendance un *retiming* légal (en particulier dans la preuve de l'algorithme de compaction).

En résumé, dans un modèle à latences quelconques (dont certaines négatives), un ordonnancement peut correspondre à la composition d'un *retiming* illégal, d'une compaction de boucle et d'un éventuel recouvrement des motifs successifs dans la limite des dates de départ des opérations. Ce qui signifie qu'une approche utilisant des *retimings* légaux uniquement ne peut pas exprimer tous les ordonnancements possibles. Notons toutefois que la plupart des travaux sur le pipeline logiciel utilisent un modèle proche de notre modèle non pipeliné, dans lequel la latence est égale au temps d'exécution de l'opération source, c'est-à-dire toujours positive.

2.3.3 Sélection d'un *retiming* pour la compaction de boucle

Reprenons le problème à l'envers : dans un modèle à latences positives, tout ordonnancement cyclique est la composition d'un *retiming* légal et d'une compaction avec recouvrement. Inversement, tout *retiming* légal peut être suivi d'une compaction produisant un ordonnancement cyclique. L'idée du pipeline logiciel décomposé est alors immédiate : nous pouvons tenter de résoudre le problème de l'ordonnancement cyclique d'une boucle en deux temps, dans un premier temps par une détermination d'un *retiming* approprié, dans un second temps par une compaction du graphe résultant. La question qui reste en suspens dans cette méthode est alors : comment sélectionner un « bon » *retiming* de manière à produire la meilleure compaction possible ? Afin de répondre à cette question, considérons tout d'abord la stratégie mise en œuvre par les différents algorithmes de décalage/ordonnancement.

L'*enhanced software pipelining* et ses extensions [80], le *circular software pipelining* [61] et le *rotation software pipelining* [13] utilisent une approche similaire : ils compactent la boucle, puis décalent d'une itération en arrière (resp. en avant) les sommets apparaissant au début (resp. à la fin) du motif servant de corps à la boucle. En d'autres termes, les candidats pour un décalage en arrière (*retiming* négatif) sont les sources de $A(G)$ et les candidats pour un décalage en avant (*retiming* positif) sont les puits de $A(G)$. Cette « rotation » est effectuée tant que l'ordonnancement s'améliore, le problème est qu'aucune garantie n'est donnée pour une telle approche.

Dans le cas du pipeline logiciel décomposé, le principe est radicalement différent. Bien qu'il s'agisse toujours de déterminer un décalage et une compaction, l'algorithme n'est plus un processus itératif utilisant en alternance les deux techniques. Le décalage (ou *retiming* r) est choisi en une seule étape, suivant un objectif précisément défini mathématiquement, puis la boucle est compactée en considérant $A(G_r)$. Puisqu'un *retiming* r change le poids des dépendances et en particulier les dépendances de poids nul (donc $A(G_r)$), le but est de déterminer le *retiming* r produisant le graphe $A(G_r)$ le plus susceptible d'être compacté efficacement. Dans [49] cette approche est formalisée[4], Gasperoni et Schwiegelshohn concluent alors en suggérant qu'il serait possible d'essayer de « couper autant d'arcs de G que possible [...] de manière à "maximiser le parallélisme" dans le graphe dérivé G' »[5] (où G' est notre $A(G_r)$, le graphe acyclique pour la compaction). Les objectifs principaux lors de la détermination du *retiming* le plus approprié sont alors les suivants :

- minimiser la « durée » du chemin critique pour la compaction $\Phi(G)$, puisqu'elle est étroitement liée à la performance de la compaction. Comme nous l'avons rappelé en section 2.3.1, $\Phi(G)$ est une borne inférieure à l'*intervalle de lancement* trouvé par compaction (sans recouvrement). De plus, en cas d'utilisation d'un ordonnancement par liste lors de la compaction, la réduction de $\Phi(G)$ entraîne également la réduction de la borne supérieure à la performance de notre ordonnancement cyclique ;
- minimiser le nombre d'arcs du graphe acyclique $A(G_r)$, de manière à réduire le nombre de contraintes à respecter lors de la compaction. Bien que n'offrant pas de meilleure garantie théorique, cette idée suit les intuitions formulées dans [49, 10] : moins l'ordonnanceur a de contraintes à respecter, plus il dispose de libertés pour exploiter les ressources disponibles.

Jusqu'alors, quasiment tous les efforts ont été consacrés à la réalisation du premier objectif. Dans [48] et [105], la boucle est d'abord ordonnancée sous l'hypothèse de ressources illimitées. A partir de cet ordonnancement initial, les deux articles proposent respectivement soit de « couper » des arcs du graphe de dépendance, soit de calculer des numéros de « ligne » et de « colonne » pour les instructions du graphe. Bien que différentes en apparence, ces deux techniques reviennent à exprimer un *retiming* légal r auquel correspond l'ordonnancement cyclique obtenu en ressources infinies. Il suffit ensuite d'appliquer une compaction de boucle, cette fois en ressources limitées, sur le graphe $A(G_r)$ pour obtenir notre ordonnancement final. Sauf que cette fois-ci le chemin critique de $A(G_r)$ n'est plus arbitrairement long comme c'était le cas pour $A(G)$. Par définition de r, $A(G_r)$ respecte les dépendances exprimées par le motif de l'ordonnancement en ressources infinies, et il est possible de prouver que, dans un modèle similaire à notre modèle non pipeliné, la « durée » de ses chemins critiques est bornée par $\lceil \lambda_\infty \rceil + \max_{u \in V}(t_\Delta(u)) - 1$. Dans [10], le *retiming* r est choisi directement, sans ordonnancement préalable, de manière à minimiser le chemin critique pour la compaction de $A(G_r)$. Ceci se fait en utilisant l'algorithme de Leiserson et Saxe (voir la section 1.2.3) pour la minimisation de la période d'horloge. Il ne reste ensuite plus qu'à compacter $A(G_r)$. Ici encore, grâce au choix du *retiming*, et dans notre modèle non pipeliné, il est possible de montrer que la

[4] Bien que cela ne soit pas fait en termes de *retiming*, le résultat est équivalent : un *retiming* est trouvé et suivi d'une compaction.

[5] *cut as many edges from G as allowed [...] so as to "maximize the parallelism" in the derived graph G'.*

« durée » d'un chemin critique de $A(G_r)$ est bornée par $\lceil \lambda_\infty \rceil + \max_{u \in V}(t_\Delta(u)) - 1$. Dans tous les cas, en utilisant un ordonnancement par liste pour la compaction, la technique de Graham nous permet d'obtenir les bornes suivantes :

$$\begin{aligned} \lambda_\sigma &\leq \lambda_p + \frac{p-1}{p}(\lceil \lambda_\infty \rceil + \max_{u \in V}(t_\Delta(u)) - 1) \\ &\leq (2 - \frac{1}{p})\lceil \lambda_p \rceil + \frac{p-1}{p}(\max_{u \in V}(t_\Delta(u)) - 1) \end{aligned} \tag{2.9}$$

Ainsi, toutes ces techniques mènent à une garantie similaire. Contrairement au cas de la compaction seule, le chemin critique dans $A(G_r)$ est maintenant relié à λ_p. La performance n'est plus arbitrairement mauvaise et il est possible de garantir son écart par rapport à l'optimal. Il est à noter que dans le cas de ressources pipelinées de temps de réservation $t_r = 1$ avec des latences toujours sur les noeuds, l'heuristique de Gasperoni et Schwiegelshohn ([48]) abaisse encore cette borne en éliminant le terme $\max_{u \in V}(t_\Delta(u))$. Ceci est dû à la manière très particulière d'ordonnancer qu'ils utilisent, de façon à garantir un bon recouvrement (au prix d'une contrainte sur l'ordonnanceur). Nous ne présentons pas les détails théoriques de ce cas ici, mais nous en discutons plus longuement dans la section 2.5, lors de la présentation des résultats des expérimentations.

Cependant, bien que ces techniques atteignent toutes la même performance théorique, le second objectif n'est envisagé que par Calland, Darte et Robert. Le but est ici d'essayer de trouver le *retiming* minimisant le nombre de contraintes pour la compaction. Dans [10], ils proposent une formulation par programmation linéaire en nombre entiers (bien que de résolution polynomiale) permettant de choisir parmi les *retimings* minimisant le chemin critique celui produisant le moins de contraintes pour la compaction. Par la suite, nous montrons qu'un algorithme de graphe permet de résoudre ce problème, comme ils le suspectèrent. De plus, nous verrons que la minimisation des contraintes pour la compaction peut être soit vue comme un problème indépendant, soit combinée à la minimisation du chemin critique, soit encore combinée à des contraintes de chemin plus faibles. Auparavant, nous reformulons l'algorithme de Leiserson et Saxe en l'étendant à notre modèle à latence sur les arcs, et nous revenons à notre exemple afin d'illustrer notre propos. Il est à noter que l'approche de Gasperoni et Schwiegelshohn non pas basée sur le *retiming* mais sur l'ordonnancement en ressources infinies est difficilement combinable à d'autres objectifs : le contrôle sur le *retiming* produit demanderait un contrôle sur l'ordonnancement produit, ce qui semble impossible.

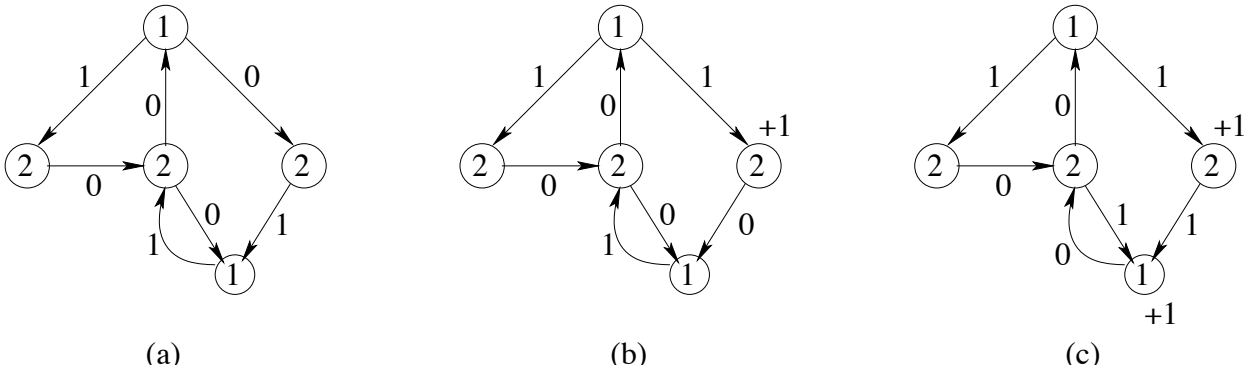


FIG. 2.5 – Graphe original (a) et graphes obtenus après minimisation de la période d'horloge (b) et du nombre d'arcs de poids nul (c).

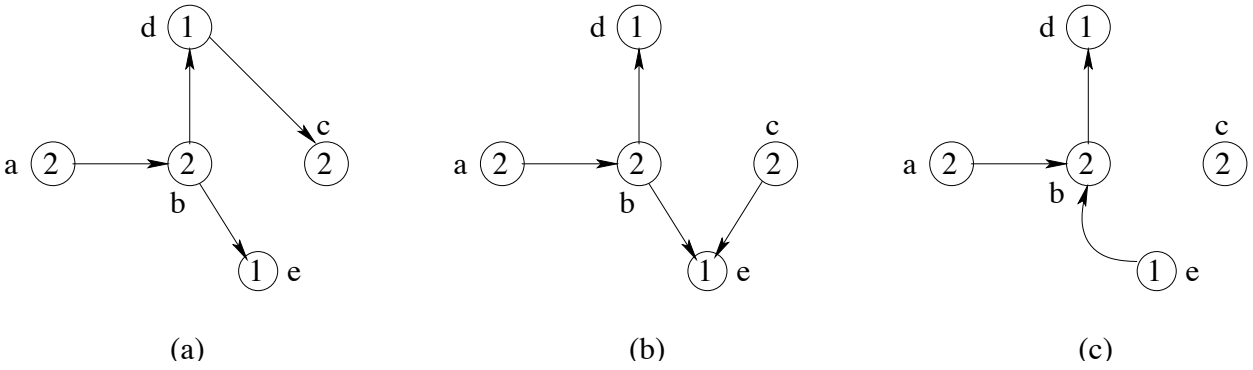


FIG. 2.6 – Les trois graphes acycliques $A(G)$ considérés pour la compaction.

Exemple (suite) Nous revenons à notre exemple (toujours dans une situation simple dans laquelle la latence est sur les noeuds). En examinant chaque circuit du graphe, nous trouvons le ratio latence sur poids minimal, qui est également l'*intervalle de lancement* de l'ordonnancement en ressources infinies :

$$\lambda_\infty = \frac{5}{1} = 5$$

L'algorithme de minimisation de la période d'horloge nous permet de trouver un *retiming* r tel que $\Phi(G_r) = \Phi_{opt}(G) = 5$. La figure 2.5 montre le graphe original (a), le graphe après un *retiming* minimisant la période d'horloge (avec l'algorithme de Leiserson et Saxe) (b) et le graphe après un

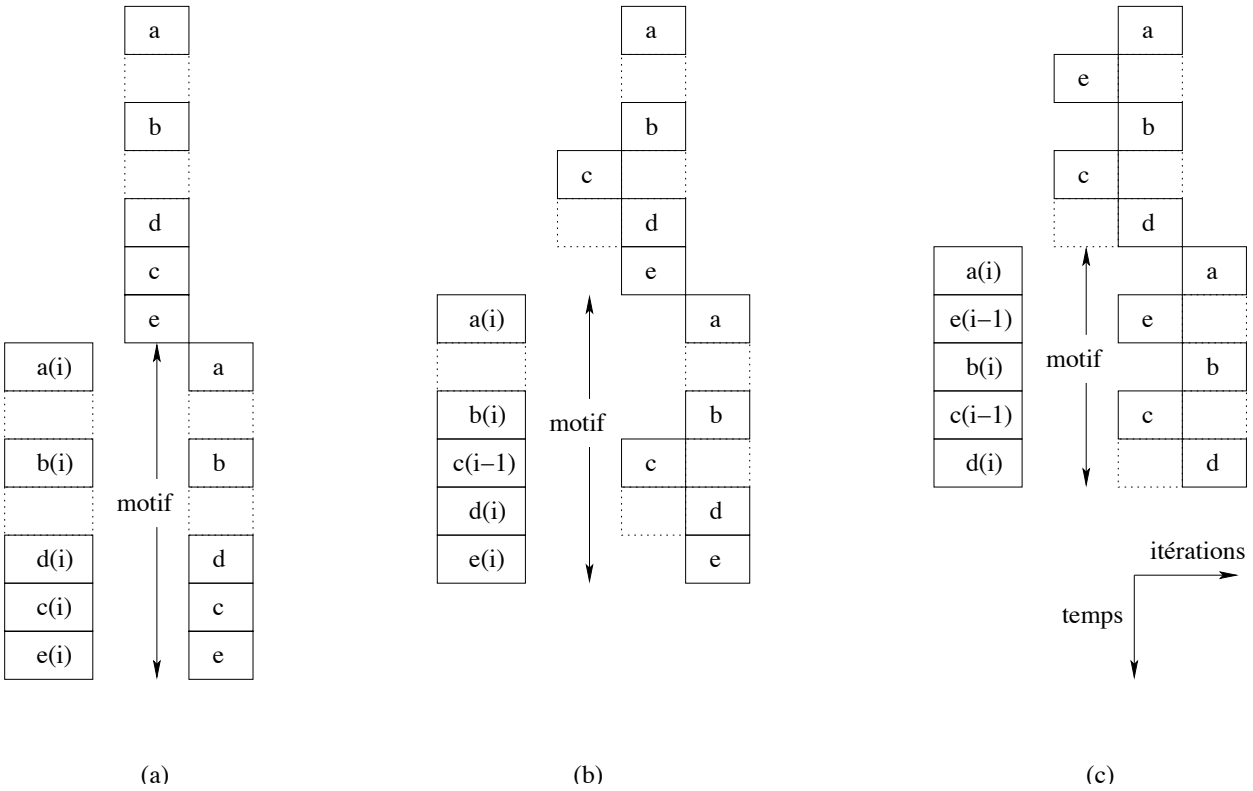


FIG. 2.7 – Les trois compactions correspondantes.

retiming minimisant le nombre d'arcs de poids nul (c). L'algorithme de Leiserson et Saxe est rappelé en section 1.2.3, tandis que l'algorithme de minimisation du nombre d'arcs de poids nul fait l'objet de la section 2.4. Les trois graphes acycliques pris en compte pour la compaction sont donnés en figure 2.6.

Dans le second et le troisième graphe (figures 2.6 (b) et (c)), le chemin critique dans $A(G)$ est réduit à 5, ce qui laisse augurer une meilleure compaction. Dans le troisième (figure 2.6 (c)), une liberté supplémentaire est donnée (l'instruction c ne dépend plus d'une autre et peut être utilisée à tout endroit de l'ordonnancement pour remplir un trou). Les compactions résultantes de ces trois graphes sont données en figure 2.7, avec pour *intervalles de lancement* respectifs 7, 6, et 5 (le temps s'écoule verticalement du haut vers le bas).

2.3.4 L'algorithme de Leiserson et Saxe étendu

L'algorithme de Leiserson et Saxe, présenté en section 1.2.3, permet de déterminer un *retiming* minimisant la période d'horloge d'un graphe. Cette période d'horloge est identique (même définition) au chemin critique pour la compaction dans notre modèle non pipeliné, c'est-à-dire avec des latences égales au temps de calcul du sommet source et où une seule information de temps d'exécution sur les sommets est suffisante. Nous proposons ici une extension de cet algorithme à notre modèle pipeliné dans lequel le graphe possède une latence sur ses arcs et où la « durée » du chemin critique (donc la période d'horloge) est définie comme la somme des latences plus le t_Δ de la dernière tâche d'un chemin de $A(G)$. Bien que ce cas ressemble au cas des délais de propagation non uniformes proposé dans [70], il est différent car il ne distingue pas les délais de propagation d'un arc à un autre, mais bien d'un sommet à un autre. Par ailleurs, dans cette présentation, nous nous limitons à un algorithme de graphe ne faisant pas appel à la programmation linéaire.

Parmi les composantes de l'algorithme de Leiserson et Saxe, nous ne modifions que les algorithmes CP (pour calculer la période d'horloge) et WD (pour calculer les matrices W et D). Nous commencons par l'algorithme CP.

Algorithme 7 (*CP étendu*)

- soit $A(G)$ le sous-graphe des arcs de poids nul de G ;
- par correction de G, $A(G)$ est acyclique, calculer une extension linéaire (ordre topologique) de $A(G)$;
- parcourir les sommets de $A(G)$ dans l'ordre topologique, pour chaque sommet v visité calculer la quantité suivante :
  - s'il n'y a pas d'arc entrant dans v, poser $\Delta(v) \leftarrow 0$;
  - sinon poser $\Delta(v) \leftarrow \max\{l(e) + \Delta(u) \mid e = (u, v) \in E \text{ et } d(e) = 0\}$;
- pour tout sommet $v \in V$, poser $\Delta(v) \leftarrow \Delta(v) + t_\Delta(v)$;
- la période d'horloge $\Phi(G)$ est $\max_{v \in V} \Delta(v)$.

Le fonctionnement est le même que pour l'algorithme CP original, mais le calcul est adapté à la définition de la « durée » du chemin critique. Notons que dans notre cas nous commençons par calculer pour chaque sommet la latence maximale de tout chemin y arrivant. La quantité t_Δ étant fixe pour un sommet donné, il suffit de l'ajouter à la fin. Nous poursuivons par l'algorithme WD étendu.

Algorithme 8 (*WD étendu*)

- pondérer chaque arc $e \in E$ par le vecteur $(d(e), -l(e))$;
- en utilisant cette pondération, calculer le poids du plus court chemin joignant toute paire de sommets à l'aide d'un algorithme standard (Floyd-Warshall par exemple) en utilisant l'ordre lexicographique ;
- pour chaque plus court chemin (x, y) joignant deux sommets u et v, poser $W(u, v) \leftarrow x$ et $D(u, v) \leftarrow t_\Delta(v) - y$.

Là encore, même fonctionnement adapté à la nouvelle définition. La suite est identique à celle présentée en section 1.2.3. Dans un premier temps il convient d'être capable de vérifier qu'une période d'horloge donnée est réalisable par *retiming*. Pour cela, nous disposons du théorème 1 qui se traduit sous la forme de l'algorithme $OPT1$. Nous obtenons alors le *retiming* produisant la période d'horloge minimale que nous notons $\Phi_{opt}(G)$.

2.3.5 L'algorithme de Calland, Darte et Robert

L'algorithme de Calland, Darte et Robert (CDR) est le premier avec [14] à proposer une approche du pipeline logiciel décomposé à base de *retiming*. Jusqu'alors, les algorithmes de pipeline logiciel décomposé procédaient par « découpage » d'arcs dans le graphe de dépendance, ces arcs devenant alors des arcs portés par la boucle, et le graphe résultant correspondant à $A(G_r)$. Ce découpage est bien sûr équivalent à un *retiming* et un ordonnancement acyclique ; dans [97] les valeurs de *retiming* et les dates d'ordonnancement sont respectivement nommées numéros de colonne et numéros de ligne (en référence à la représentation classique en cascade d'une boucle pipelinée). Cependant, l'arrivée du *retiming* est intéressante pour deux raisons :

- il utilise l'algorithme de Leiserson et Saxe, qui est déjà connu, et qui permet, entre autres, d'exprimer les contraintes sur le chemin critique de façon très simple à l'aide d'arcs additionnels dans le graphe ;
- les contraintes de poids sur certains arcs ou chemins s'intégrant dans le graphe lui-même à l'aide de nouveaux arcs, il est possible de combiner les objectifs, en particulier, comme nous allons le voir, nous pouvons minimiser le nombre d'arcs de poids nul sous contrainte de période d'horloge.

Ainsi, bien qu'apportant une garantie de performance équivalente à celle de Gasperoni et Schwiegelshohn [48], l'algorithme CDR montre que l'on peut faire plus que simplement minimiser la durée du chemin critique grâce à la flexibilité des techniques de *retiming*. Bien que formulé légèrement différemment dans [10], l'algorithme CDR revient à effectuer les deux étapes suivantes :

Algorithme 9 (*Algorithme CDR (version condensée)*)

- trouver un *retiming* r minimisant la période d'horloge de G, avec l'algorithme de Leiserson et Saxe ;
- utiliser l'algorithme de compaction de boucle sur G_r pour trouver un ordonnancement cyclique de G.

Le point remarquable est la relation entre $\Phi_{opt}(G)$ et λ_∞ montrée par Calland, Darte et Robert et fournissant une garantie de performance à cette technique. Nous la reformulons ici en l'adaptant à notre modèle étendu :

Lemme 4 (performance de la période d'horloge minimale) *Soit un graphe de dépendance G, nous avons :*

$$\lambda_\infty \leq \Phi_{opt}(G) \leq \lambda_\infty + \max_{u \in V} t_\Delta(u) - 1$$

Preuve Tout d'abord, appliquons l'algorithme CDR à G avec une infinité de ressources. Pour ceci, nous trouvons grâce à l'algorithme de Leiserson et Saxe un *retiming* r tel que $\Phi(G_r) = \Phi_{opt}(G)$. La compaction est alors appliquée à $A(G_r)$, en ressources infinies, et pour toute opération u nous pouvons choisir $\sigma_a(u) = \max\{l(p) | p \text{ est un chemin menant à } u\}$. L'*intervalle de lancement* de la boucle ainsi compactée est par définition $\lambda_\sigma = \max_{u \in V}(\sigma_a(u) + t_\Delta(u)) = \Phi(G_r) = \Phi_{opt}(G)$. L'ordonnancement final est alors $\sigma(v,k) = \sigma_a(v) + \Phi_{opt}(G)(k + r(v))$. Puisque λ_∞ est par définition le plus petit *intervalle de lancement* pour $p = \infty$, alors $\lambda_\infty \leq \Phi_{opt}(G)$.

Considérons maintenant un ordonnancement optimal en ressources infinies, $\sigma_{opt}(v,k) = \lambda_\infty k + a_{opt}(v)$. Soit $a'_{opt}(v) = a_{opt}(v) \mod \lambda_\infty$ et $r(v) = \lfloor a_{opt}(v)/\lambda_\infty \rfloor$ pour tout sommet v de G. Comme tout ordonnancement, σ_{opt} respecte les contraintes de ressources, et de façon analogue à l'équation 2.6 de la section 2.3.2, nous obtenons :

$$\lambda_\infty(d(e) + r(v) - r(u)) \quad \geq \quad l(e) + a'_{opt}(u) - a'_{opt}(v)$$

Donc tous les arcs tels que $d(e) + r(v) - r(u) = 0$, c'est-à-dire par définition tous les arcs de $A(G_r)$, vérifient :

$$a'_{opt}(u) + l(e) \quad \leq \quad a'_{opt}(v)$$

Donc pour tout chemin critique $p = \{u_1, \ldots, u_n\}$ de $A(G_r)$, en sommant les $(n-1)$ inégalités sur les arcs (u_i, u_{i+1}) où $i \in \{1, \ldots, n-1\}$, nous obtenons :

$$\begin{array}{rrcl}
 & \sum_{i=1}^{n-1} l(u_i, u_{i+1}) + a'_{opt}(u_1) & \leq & a'_{opt}(u_n) \\
\iff & \sum_{i=1}^{n-1} l(u_i, u_{i+1}) + t_\Delta(u_n) - t_\Delta(u_n) + a'_{opt}(u_1) & \leq & a'_{opt}(u_n) \\
\iff & \Phi(G_r) - t_\Delta(u_n) + a'_{opt}(u_1) & \leq & a'_{opt}(u_n) \\
\Rightarrow & \Phi(G_r) & \leq & a'_{opt}(u_n) + t_\Delta(u_n) \\
\Rightarrow & \Phi_{opt}(G) & \leq & (\lambda_\infty - 1) + \max_{u \in V} t_\Delta(u)
\end{array}$$

puisque $a'_{opt}(v) = a_{opt}(v) \mod \lambda_\infty \Rightarrow 0 \leq a'_{opt}(v) < \lambda_\infty$. □

Ainsi, après la minimisation de la période d'horloge par l'algorithme de Leiserson et Saxe, nous pouvons remplacer $\Phi(G)$ dans les équations 2.4 et 2.5 par $\Phi_{opt}(G)$. Nous retrouvons alors l'inéquation 2.9 dans le cas non pipeliné, et dans le cas pipeliné nous obtenons :

$$\begin{array}{rcl}
\lambda_\sigma & \leq & \lambda_p + \left(1 - \dfrac{1}{p\,(\max_{u \in V} t_\Delta(u))}\right)(\lambda_\infty + \max_{u \in V} t_\Delta(u) - 1) \\
 & \leq & \left(2 - \dfrac{1}{p\,(\max_{u \in V} t_\Delta(u))}\right)\lambda_p + \left(\dfrac{p\,(\max_{u \in V} t_\Delta(u)) - 1}{p\,(\max_{u \in V} t_\Delta(u))}\right)(\max_{u \in V} t_\Delta(u) - 1)
\end{array}$$

Autrement dit, nous disposons d'un résultat qui est au pire environ à un facteur 2 de l'optimal (en négligeant la partie $(\max_{u \in V} t_\Delta(u) - 1)$ qui est constante). Avec ce résultat de Gasperoni et Schwiegelshohn, reformulé en termes de *retiming* par Calland, Darte et Robert et étendu ici à un

modèle plus général, le pipeline logiciel décomposé est, à notre connaissance, la seule technique de pipeline logiciel possédant une garantie théorique de performance. Cette seule caractéristique en fait un outil de recherche très intéressant. Néanmoins, nous allons par la suite nous consacrer d'une part à la réalisation d'autres objectifs par *retiming*, comme la minimisation du nombre de contraintes pour la compaction, et d'autre part étudier l'efficacité absolue de la technique par des expérimentations.

2.4 Minimisation du nombre de contraintes pour la compaction

Dans cette section, nous présentons le problème de la recherche d'un *retiming* permettant de minimiser le nombre total de contraintes restant lors de l'application d'une compaction de boucle. Formulée en termes de *retiming*, cette recherche devient un problème théorique de graphe : comment trouver un *retiming* tel que le nombre d'arcs de poids nul dans le graphe transformé soit minimal. Nous étudions ici cette minimisation comme un élément indépendant (qui pourrait éventuellement trouver son application dans d'autres problèmes d'optimisation). Nous ne reviendrons au lien effectif avec le pipeline logiciel et à ses avantages et inconvénients vis-à-vis de la compaction de boucle que dans les sections ultérieures (essentiellement les sections 2.5 and 2.6).

Bien sûr, il existe des cas pour lesquels toutes les dépendances peuvent être éliminées : il est possible d'employer un simple algorithme polynomial afin de trouver un *retiming* parvenant à ce résultat ou de prouver qu'il n'existe pas de tel *retiming* (nous présentons brièvement ce cas particulier en section 2.4.1). Cependant, il n'est en général pas possible d'éliminer toutes les dépendances grâce à cette transformation, c'est pourquoi nous devons recourir à un algorithme plus sophistiqué (quoique néanmoins polynomial et efficace) afin de minimiser le nombre d'arcs de poids nul de notre graphe. Cet algorithme est exposé en détails en section 2.4.2 et nous l'appliquons à notre exemple principal afin d'en illustrer le fonctionnement. Puis en section 2.4.3, nous étendons cet algorithme afin de permettre l'ajout de contraintes sur le *retiming* cherché, par exemple pour imposer une limite supérieure à la période d'horloge du graphe résultant – cette limite supérieure devant être bien sûr réalisable, et pouvant être par exemple la période d'horloge minimale du graphe. Ceci nous permettra alors de combiner nos deux objectifs initiaux proposés en section 2.3.3 : trouver un *retiming* minimisant d'une part le chemin critique et d'autre part le nombre total de contraintes pour la compaction.

2.4.1 Un cas particulier : le corps totalement parallèle

Lorsque toutes les dépendances peuvent être transformées en dépendances portées par la boucle, toutes les opérations sont alors indépendantes à l'intérieur du corps de la boucle. Dans ce cas, la compaction est alors réduite au problème d'ordonnancer les opérations sans contraintes de précédence qui, bien que NP-complet également, est un problème plus simple : des heuristiques garanties offrant une performance meilleure que l'ordonnancement par liste existent. Plus formellement, nous dirons que le corps d'une boucle est totalement parallèle lorsque son graphe de dépendance satisfait la condition suivante :

$$\forall e \in E,\ d(e) \geq 1$$

En d'autres termes, si nous pouvons trouver un *retiming* r tel que, pour tout arc $e \in E, d_r(e) \geq 1$, nous produisons une boucle à corps totalement parallèle. L'algorithme 2 de *retiming* sous contraintes (présenté en section 1.2.2) permet de trouver un tel *retiming* ou de prouver qu'il n'en existe pas. Cet algorithme a une complexité polynomiale en $O(|V||E|)$.

Nous pouvons également remarquer que si le graphe est acyclique, il existe toujours un *retiming* rendant le corps de la boucle totalement parallèle puisqu'il n'y a pas de circuit du tout. Dans ce cas, l'algorithme pourrait même être simplifié en un simple parcours topologique du graphe menant à une complexité en $O(|V| + |E|)$ au lieu du $O(|V||E|)$ de l'algorithme 2.

2.4.2 Cas général : minimisation du nombre d'arcs de poids nul

Puisqu'il n'est pas toujours possible de rendre le corps de la boucle totalement parallèle, il nous faut trouver une autre solution afin de minimiser le nombre de contraintes pour la compaction. L'idée d'employer cette méthode afin d'améliorer l'efficacité d'une approche décomposée du pipeline logiciel est déjà présente dans [10]. Dans cet article, Calland, Darte et Robert proposent une résolution du problème par programmation linéaire en nombres entiers. Ils montrent que la matrice du programme est totalement unimodulaire, assurant ainsi la possibilité d'une résolution en temps polynomial. Notre premier but ici est d'améliorer la méthode de résolution de Calland, Darte et Robert en s'affranchissant du recours à la programmation linéaire. Nous présentons dans cette section un algorithme de graphe permettant de trouver un *retiming* pour lequel le nombre d'arcs de poids nul est minimal dans le graphe transformé (en d'autres termes, autant de dépendances que possible sont portées par la boucle après *retiming*). L'algorithme est inspiré d'un algorithme de recherche de flot de coût minimal connu sous le nom de méthode des arcs non conformes (voir [38, pp. 178-185]), et proposée par Fulkerson en 1961.

Analyse du problème

Étant donné un graphe de dépendance $G = (V, E, d)$ et un *retiming* r de G, nous souhaitons compter le nombre d'arcs e tels que $d_r(e) = 0$ dans le but de le minimiser. Pour cela, nous définissons le coût $v_r(e)$ d'un arc e de la manière suivante :

$$v_r(e) = \begin{cases} 1 & \text{si } d_r(e) = 0 \\ 0 & \text{sinon} \end{cases}$$

Et nous définissons le coût total du *retiming* r comme $V(r) = \sum_{e \in E} v_r(e)$, c'est-à-dire le nombre d'arcs de poids nul du graphe transformé. Nous dirons que r est optimal lorsque $V(r)$ est minimal c'est-à-dire lorsque r minimise le nombre d'arcs de poids nul du graphe.

Par la suite nous utiliserons la notion de flot défini sur le graphe G. Un flot est défini comme une fonction $f : E \to \mathbb{Z}$ telle que :

$$\forall v \in V, \sum_{e=(.,v) \in E} f(e) = \sum_{e=(v,.) \in E} f(e)$$

Un flot positif est un flot tel que $\forall e \in E$, $f(e) \geq 0$. Un flot f correspond à une union de cycle (un multi-cycle) C_f traversant $f(e)$ fois chaque arc du graphe sur lequel il est défini. Lorsque $f(e) > 0$ l'arc est traversé dans son sens naturel, et lorsque $f(e) < 0$ l'arc est traversé en sens inverse. Lorsqu'un flot est positif, il correspond donc à une union de circuits (un multi-circuit), le lecteur est invité a se référer à [38, p. 163] pour plus de détails. Par la suite nous n'utiliserons que des flots positifs et nous désignerons par abus un flot positif sous le terme de flot.

L'idée de l'algorithme est la suivante : tout d'abord nous devons définir un coût pour les flots que nous utilisons, de telle sorte que le coût d'un flot quelconque sur notre graphe soit toujours inférieur au coût d'un *retiming* de ce graphe. Alors, en partant du *retiming* et du flot nuls, nous allons montrer comment il est possible de les faire évoluer de manière à faire croître le coût de notre

flot tout en faisant décroître le coût de notre *retiming* jusqu'à ce que les deux coûts soient égaux. A ce point, l'optimal sera alors atteint.

Étant donné un *retiming* légal r et un flot f, nous définissons le coût $z_{f,r}(e)$ de f pour un arc e de la manière suivante[6] :

$$z_{f,r}(e) = \begin{cases} 0 & \text{si } f(e) = 0 \\ 1 - f(e)d_r(e) & \text{sinon} \end{cases}$$

et nous définissons le coût du flot f comme la somme de son coût sur tous les arcs du graphe : $\sum_{e\in E} z_{f,r}(e)$. La proposition suivante nous permet alors de montrer que, bien que le coût du flot sur un arc $z_{f,r}(e)$ soit défini pour un *retiming* donné, son coût total $\sum_{e\in E} z_{f,r}(e)$ est invariant par *retiming*. Ceci nous permet donc de noter $Z(f)$ le coût d'un flot (sans référence à un quelconque *retiming*).

Proposition 2 *Soit f un flot de G, C_f l'ensemble des arc e tels que $f(e) > 0$ et $|C_f|$ le nombre d'arcs dans C_f. Alors pour tout* retiming *légal r de G,*

$$\sum_{e\in E} z_{f,r}(e) = \sum_{e\in C_f} (1 - f(e)d_r(e)) = \sum_{e\in C_f} (1 - f(e)d(e)) = \sum_{e\in E} z_{f,0}(e).$$

Preuve

$$\begin{aligned}
\sum_{e\in E} z_{f,r}(e) &= \sum_{e\in C_f} (1 - f(e)d_r(e)) \\
&= |C_f| - \sum_{e=(u,v)\in E} f(e)(d(e) + r(v) - r(u)) \\
&= |C_f| - \sum_{e\in E} f(e)d(e) - \sum_{e=(u,v)\in E} f(e)r(v) + \sum_{e=(u,v)\in E} f(e)r(u) \\
&= |C_f| - \sum_{e\in E} f(e)d(e) - \sum_{v\in V} r(v) \underbrace{\left(\sum_{e=(.,v)\in E} f(e) - \sum_{e=(v,.)\in E} f(e) \right)}_{= 0 \text{ puisque } f \text{ est un flot}} \\
&= |C_f| - \sum_{e\in E} f(e)d(e) \\
&= \sum_{e\in E} z_{f,0}(e), \text{ le coût de } f \text{ pour le } \textit{retiming} \text{ nul.}
\end{aligned}$$

□

Ainsi, le coût d'un flot pour un *retiming* donné r est égal à son coût pour le *retiming* nul, c'est-à-dire son coût dans le graphe initial. En d'autres termes, le coût total d'un flot ne dépend pas du *retiming* du graphe.

Pour un *retiming* r et un flot positif f donnés, nous définissons pour chaque arc $e \in E$ son indice de conformité $ki(e) = v_r(e) - z_{f,r}(e)$. Il est facile de vérifier que :

- $ki(e) = 0$ lorsque l'une des condition suivante est vérifiée :
  - $f(e) = 0$ et $d_r(e) > 0$;

[6] Cette définition peut sembler mystérieuse à première vue, mais elle est déduite, après quelques manipulations dues à une fonction objective complexe, de l'analyse du problème exprimé sous la forme du programme linéaire donné dans [10].

  - $f(e) > 0$ et $d_r(e) = 0$;
  - $f(e) = 1$ et $d_r(e) = 1$.
- $ki(e) = 1$ lorsque $f(e) = 0$ et $d_r(e) = 0$;
- $ki(e) = f(e)d_r(e) - 1$ dans tous les autres cas (et alors, $ki(e) > 0$).

Ainsi, l'indice de conformité d'un arc est toujours positif. Ceci implique alors la proposition suivante, qui donne une borne inférieure au coût d'un *retiming* et une borne supérieure au coût d'un flot.

Proposition 3 *Pour tout* retiming *légal r et pour tout flot positif f, $V(r) \geq Z(f)$. Ceci implique que $V(r) \geq \min\{V(r) \mid r$ retiming *légal*$\} \geq \max\{Z(f) \mid f$ *flot positif*$\} \geq Z(f)$.*

Preuve Puisque pour tout *retiming* légal r et tout flot positif f tous les indices de conformité de tous les arcs du graphe sont positifs, nous avons $V(r) - Z(f) = \sum_{e \in E} ki(e) \geq 0$. Ainsi, puisque $V(r) \geq Z(f)$ pour tout flot positif f, nous obtenons $V(r) \geq \max\{Z(f) \mid f$ flot positif$\}$. Enfin, puisque ceci est vrai pour tout *retiming* légal, nous obtenons $\min\{V(r) \mid r$ *retiming* légal$\} \geq \max\{Z(f) \mid f$ flot positif$\}$. □

Nous obtenons finalement une condition suffisante pour qu'un *retiming* légal soit optimal.

Proposition 4 *Soit r un* retiming *légal. S'il existe un flot f tel que $\forall e \in E$, $ki(e) = 0$, alors r est optimal.*

Preuve Si r et f sont tels que $\forall e \in E$, $ki(e) = 0$, alors $V(r) = Z(f)$. Et donc, d'après la proposition 3, $V(r)$ est égal à la valeur minimale possible pour V. □

Néanmoins, la proposition 4 seule ne montre pas que cette condition est nécessaire, ni que la borne inférieure au coût d'un *retiming* peut toujours être atteinte. Il nous reste donc à montrer qu'il est toujours possible de trouver un *retiming* et un flot tels que l'indice de conformité de tous les arcs du graphe soit nul. Nous allons donc étudier dans quelles circonstances cela peut se produire, en commençant par une caractérisation des arcs du graphe en fonction de leur indice de conformité.

Caractérisation des arcs

Nous allons représenter le flot et le poids après *retiming* d'un arc par une paire $(f(e), d_r(e))$. Ainsi, en plaçant sur le plan les différents points correspondant aux différentes valeurs possibles de cette paire, nous obtenons le diagramme représenté en figure 2.8, aussi appelé diagramme de conformité. Les valeurs $f(e)$ et $d_r(e)$ pour lesquelles $ki(e) = 0$ correspondent à la ligne brisée qui traverse le diagramme. Au dessous et au dessus de cette ligne se trouvent les paires de valeurs pour lesquelles $ki(e) > 0$. Nous appelons conforme un arc e pour lequel $ki(e) = 0$, et non conforme un arc e pour lequel $ki(e) > 0$. Comme illustré sur le diagramme, nous assignons à chaque arc une couleur dépendant de la position de son couple de valeurs $(f(e), d_r(e))$ par rapport à la ligne pour laquelle $ki(e) = 0$. Ceci se traduit en simples conditions sur les valeurs $f(e)$ et $d_r(e)$:

- si $(f(e), d_r(e)) = (0, 0)$, $(0, 1)$ ou $(1, 0)$, l'arc sera noir ;
- si $f(e) = 0$ et $d_r(e) > 1$, l'arc sera incolore ;
- si $f(e) > 1$ et $d_r(e) = 0$, l'arc sera rouge ;
- dans tous les autres cas, l'arc sera vert.

Si tous les arcs sont conformes, c'est-à-dire si $ki(e) = 0$ pour tout arc e, alors l'optimal est atteint. De plus, nous pouvons remarquer que, pour tout arc conforme e, il est possible de modifier de 1 l'une ou l'autre des valeurs $d_r(e)$ ou $f(e)$ tout en gardant l'arc conforme. Par exemple, pour un arc e rouge, si $f(e)$ augmente ou diminue de 1, l'arc reste conforme (mais peut changer de couleur). De façon analogue, pour tout arc e non conforme, il est possible de modifier de 1 l'une ou l'autre de

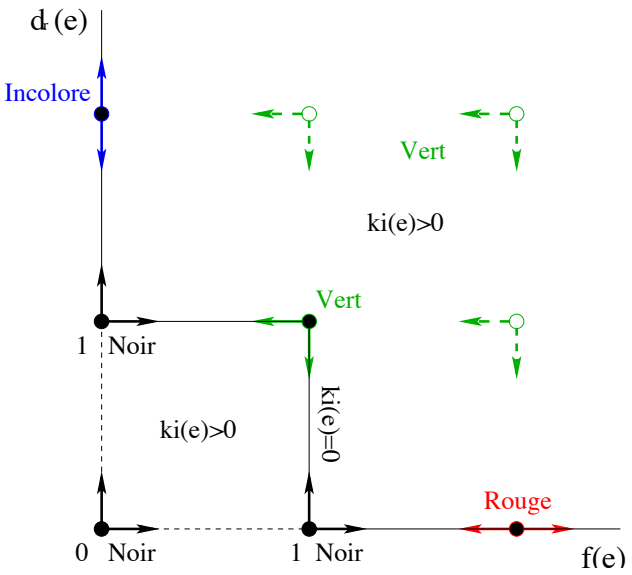


FIG. 2.8 – Diagramme de conformité et couleur des arcs.

valeurs $d_r(e)$ ou $f(e)$ afin de voir son indice de conformité décroître strictement. Par exemple, pour un arc e noir tel que $f(e) = 0$ et $d_r(e) = 0$, il est possible d'augmenter de 1 soit $f(e)$ soit $d_r(e)$ et de le rendre ainsi conforme. Nous allons donc exploiter ces possibilités afin de converger vers une solution optimale par modifications successives du *retiming* ou du flot.

L'algorithme final démarre d'une solution initiale admissible et la fait évoluer vers une solution optimale. Le *retiming* et le flot nuls sont respectivement un *retiming* légal et un flot positif de G, il forment une solution admissible initiale. Pour cette solution, $ki(e) = v_r(e)$ pour tout arc $e \in E$, ce qui signifie un indice de conformité de 1 pour les arcs de poids nul et de 0 pour les autres arcs. Remarquons que pour cette solution, tous les arcs sont soit noirs soit incolores, et seulement les arcs dont le couple $(f(e), d_r(e))$ est situé à l'origine du diagramme, soit (0,0), sont non conformes. Ceci peut permettre d'éliminer des tests dans l'algorithme si l'on part du principe que la même solution initiale est toujours choisie. Le problème est alors de faire décroître l'indice de conformité des arcs non conformes sans faire croître celui des autres arcs.

Les couleurs que nous avons assignées aux arcs vont nous permettre de réaliser notre objectif. Elles expriment le degré de liberté alloué à un arc selon sa situation :

arcs noirs $f(e)$ et $d_r(e)$ peuvent seulement augmenter ;

arcs verts $f(e)$ et $d_r(e)$ peuvent seulement diminuer ;

arcs rouges $f(e)$ peut augmenter ou diminuer, mais $d_r(e)$ doit rester inchangé ;

arcs incolores $f(e)$ doit rester inchangé, mais $d_r(e)$ peut augmenter ou diminuer.

Les flèches de la figure 2.8 montrent les directions dans lesquelles le *retiming* et le flot peuvent être modifiés.

Note : lors de l'algorithme, il sera inutile de considérer les arcs verts non conformes. En effet, si nous partons d'une solution ne contenant pas de tels arcs, et si nous pouvons diminuer l'indice de conformité des arcs non conformes noirs sans augmenter celui des autres arcs, alors nous ne créerons jamais d'autres types d'arcs non conformes. En particulier, nos arcs non conformes ne seront jamais verts. Ici encore, ceci peut permettre d'éliminer des tests dans l'algorithme.

Algorithme

Nous pouvons maintenant conclure, en utilisant le lemme des couleurs de Minty (1966, voir [38, pp. 163-165]) comme dans un algorithme des arcs non conformes classique. Nous rappelons ici le lemme de Minty.

Lemme 5 (Lemme des couleurs) *Soit $G = (V, E)$ un graphe dont les arcs sont colorés en noir, vert et rouge, certains pouvant rester incolores. Supposons qu'il existe au moins un arc noir e_0. Alors une et une seule des deux propositions suivantes est vraie :*

a) il existe un cycle contenant e_0, sans arcs incolores, ayant tous ses arcs noirs orientés dans le même sens que e_0, et tous ses arcs verts orientés dans le sens opposé.

b) il existe un co-cycle contenant e_0, sans arcs rouges, ayant tous ses arcs noirs orientés dans le même sens que e_0, et tous ses arcs verts orientés dans le sens opposé.

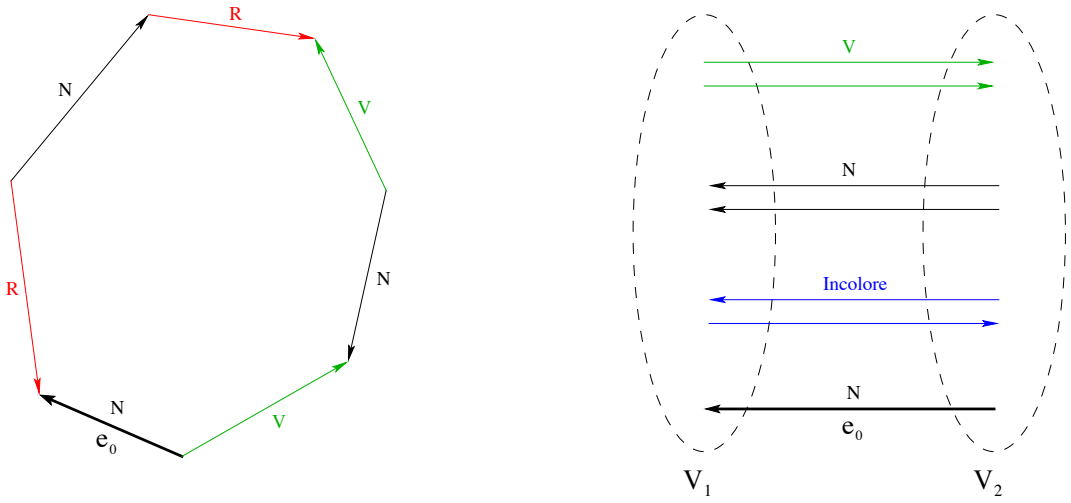


FIG. 2.9 – L'alternative du Lemme de Minty.

Preuve La preuve est constructive et fondée sur un marquage des sommets et des arcs, elle est tirée de [38]. Soit $e_0 = (u, v)$ un arc noir, nous commençons la procédure à partir de v et nous suivons les arcs incidents à v : les arcs noirs dans leur sens naturel, les arcs verts en sens inverse et les arcs rouges dans les deux sens. Les sommets et les arcs parcourus sont marqués afin de ne pas passer deux fois par le même. Si ce processus atteint u, alors le cycle désiré est trouvé : par définition de la procédure de parcours, il répond aux conditions du lemme. Sinon le processus s'arrête à un point où tous les sommets marqués n'ont plus d'arcs sortants noirs, d'arcs entrants verts, ou d'arcs rouges incidents qui ne soient pas marqués. Le marquage partitionne alors les sommets en deux sous-ensembles, l'ensemble V_1 des sommets marqués par la procédure, et $V_2 = V \setminus V_1$. Par construction tous les arcs de V_1 à V_2 sont soit verts soit incolores, et tous les arcs de V_2 à V_1 sont soit noirs soit incolores. Ces arcs entre V_1 et V_2 forment le co-cycle cherché. □

L'algorithme suit directement. Il utilise le lemme de Minty pour rapprocher à chaque étape le coût du flot de celui du *retiming*. Grâce à ce lemme, nous sommes sûrs que la progression est toujours possible.

Algorithme 10 (*Minimisation du nombre d'arcs de poids nul*)

1. démarrer avec $r(v) = 0$ pour tous les sommets $v \in V$ et $f(e) = 0$ pour tous les arcs $e \in E$;
2. colorier les arcs comme expliqué précédemment :
   - en noir si $(f(e), d_r(e)) = (0,0)$, $(0,1)$ ou $(1,0)$;
   - en incolore si $f(e) = 0$ et $d_r(e) > 1$;
   - en rouge si $f(e) > 1$ et $d_r(e) = 0$;
   - en vert dans tous les autres cas.
3. si $\forall e \in E, ki(e) = 0$, alors fin : r est un *retiming* optimal ;
4. sinon, choisir un arc non conforme $e_0 = (u, v)$ et appliquer la procédure de recherche du lemme de Minty :
   (a) si un cycle est trouvé, alors ajouter 1 (respectivement soustraire 1) au flot de tous les arcs du cycle orientés dans le sens de e_0 (respectivement orientés dans le sens inverse de e_0) ;
   (b) si un co-cycle est trouvé, il détermine une partition des sommets en deux sous-ensembles. Ajouter 1 au *retiming* de tout sommet appartenant au même sous-ensemble que v, le sommet terminal de e_0.
5. retourner à l'étape 3.

Preuve Par définition de la couleur des arcs qui exprime quels changements il est possible d'appliquer au flot ou au poids d'un arc, il est simple de vérifier qu'à chaque étape de l'algorithme l'arc non conforme e_0 choisi voit son flot ou son poids changer et devient conforme. De plus, toujours à cause de la coloration des arcs, du cycle ou du co-cycle trouvé et des changement appliqués en conséquence, il est simple de vérifier que tous les arcs conformes avant l'étape 4 de l'algorithme restent conformes après cette même étape, et que le flot reste positif et le *retiming* légal. Donc, à chaque application du lemme de Minty, au moins un arc devient conforme alors qu'il ne l'était pas. En répétant la procédure (coloriage puis changement du flot ou du *retiming*) au plus $|E|$ fois, tous les arcs deviennent conformes et le *retiming* est optimal. □

Complexité

La recherche du cycle ou du co-cycle du lemme de Minty peut être faite par une procédure de marquage en $O(|E| + |V|)$ (soit $O(|E|)$ si on suppose le graphe connexe, dans le cas contraire on peut appliquer l'algorithme à chaque composante connexe). À chaque étape, au moins un arc non conforme devient conforme, le nombre total d'étapes est donc borné par le nombre total d'arcs de poids nul dans le graphe de départ, lui-même borné par le nombre total d'arcs du graphe. La complexité totale de l'algorithme est donc $O(|E|^2)$.

Note : pour la réalisation pratique, l'algorithme pourra être optimisé afin de traiter séparément chaque composante fortement connexe du graphe de façon indépendante. En effet, une fois un *retiming* trouvé sur chacune des composantes fortement connexes, nous pouvons considérer le graphe acyclique des composantes fortement connexes et lui appliquer l'algorithme de la section 2.4.1. Ceci permet de définir une valeur à ajouter au *retiming* de tous les sommets de chaque composante fortement connexe rendant strictement positif le poids de tous les arcs entre deux composantes distinctes. Précisons que dans le cas d'un graphe acyclique, l'algorithme de la section 2.4.1 se réduit à un simple parcours topologique du graphe. Ainsi, la complexité quadratique ne s'applique qu'aux composantes fortement connexes.

Exemple (suite) Nous illustrons maintenant le fonctionnement de l'algorithme de minimisation du nombre d'arcs de poids nul sur notre exemple principal. Le graphe de dépendance de cet exemple est représenté sur la figure 2.3. Les différentes étapes de l'algorithme sont décrites pas à pas par la figure 2.10. Ces étapes sont les suivantes :

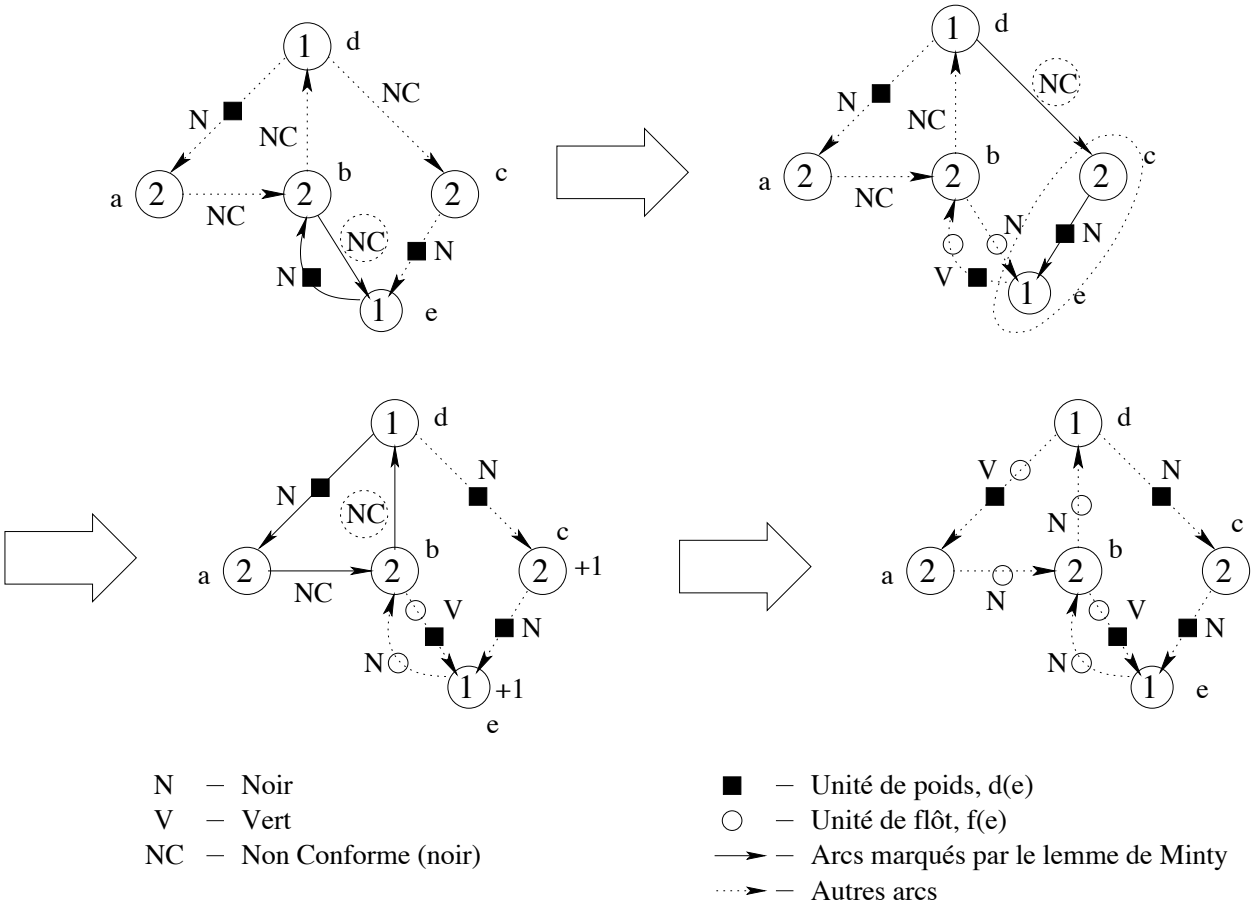


FIG. 2.10 – Les étapes successives de la minimisation du nombre d'arcs de poids nul.

- au début de l'algorithme, tous les arcs sont noirs (puisque le *retiming* et le flot sont tous deux égaux à zéro) et tous les arcs de poids nul sont non conformes (marqués NC sur la figure 2.10) ;
- nous choisissons, par exemple, l'arc non conforme de b à e, marqué par un NC encerclé sur le premier graphe de la figure 2.10. Nous suivons cet arc noir et nous trouvons un circuit (b, e, b). Nous changeons le flot comme décrit dans l'algorithme (les cercles blancs dans le second graphe représentent chacun une unité de flot) et la couleur des arcs. L'arc de b à e reste noir mais devient conforme, tandis que l'arc de e à b devient vert ;
- nous choisissons un autre arc non conforme, par exemple l'arc de d à c. Le processus de marquage marque c et e puis s'arrête. Ceci définit deux ensembles $\{a, b, d\}$ et $\{c, e\}$. Nous augmentons donc le *retiming* pour c et e. Nous changeons le poids et la couleur des arcs en conséquence (voir le troisième graphe) ;
- une fois de plus, nous choisissons un autre arc non conforme, par exemple l'arc de b à d. Nous trouvons le circuit (b, d, a, b) et nous changeons le flot en conséquence. Les deux arcs non conformes du circuit (b, d, a, b) deviennent alors conformes ;
- tous les arcs sont maintenant conformes, le *retiming* est optimal : $r(c) = r(e) = 1$ et toutes

les autres valeurs sont égales à 0. Nous retrouvons le troisième graphe de la figure 2.5 comme promis.

Notons qu'à chaque étape nous avons eu à choisir un arc non conforme : faire un autre choix que celui décrit à chacune de ces étapes aurait pu résulter en un nombre d'étapes différent et éventuellement en une solution différente. Cependant, comme nous l'avons prouvé, le résultat aurait été tout autant optimal, en d'autre termes nous aurions obtenu le même nombre d'arcs de poids nul. Notons également que sur cet exemple, la période d'horloge reste inchangée par la minimisation, il n'est pas nécessaire d'utiliser la technique présenté en section 2.4.3 pour la maintenir minimale.

2.4.3 Prise en compte de la période d'horloge

Il convient à présent d'étendre l'algorithme présenté en section 2.4.2 afin de lui donner une utilité pratique dans le cadre du pipeline logiciel décomposé. Nous voulons être à même de minimiser le nombre d'arcs de poids nul d'un graphe tout en respectant une contrainte sur la période d'horloge du résultat. Replacé dans son contexte, ceci a pour but de minimiser le nombre de contraintes pour la compaction sans augmenter la longueur du chemin critique (qui est une borne inférieure à la lenteur du résultat). En termes de graphe, nous souhaitons, étant donné un graphe de dépendance $G = (V, E)$, trouver un *retiming* de G tel que la période d'horloge de G_r soit inférieure à une constante donnée – qui doit être tout de même une période d'horloge réalisable – et de telle sorte que G_r ait aussi peu d'arcs de poids nul que possible. Rappelons que, dans notre modèle pipeliné, la période d'horloge $\Phi(G)$ d'un graphe G est définie comme la « durée » maximale d'un chemin de poids nul :

$$\Phi(G) = \max\{l(p) + t_\Delta(u_n) \mid d(p) = 0,\ p = \{u_1, \ldots, u_n\} \text{ chemin de } G\}$$

Arcs d'horloge

De façon analogue à la technique de Leiserson et Saxe présentée en section 1.2.3, nous ajoutons une nouvelle contrainte entre toute paire de sommets reliés par un chemin dont le délai est supérieur à la période d'horloge désirée Φ, car un tel chemin doit contenir au minimum un arc de poids strictement positif :

$$\forall (u, v) \in V^2,\ D(u, v) > \Phi \Rightarrow r(v) - r(u) + W(u, v) \geq 1 \tag{2.10}$$

où $D(u, v)$ et $W(u, v)$ sont calculés par l'algorithme WD étendu (voir section 2.3.4).

Comme l'illustre l'algorithme 2 de *retiming* sous contrainte, ceci correspond à l'ajout d'arcs de poids $W(u, v) - 1$ entre chaque paire (u, v) de sommets tels que $D(u, v) > \Phi$. Nous définissons E_{clock} l'ensemble de ces nouveaux arcs et $E' = E \cup E_{\text{clock}}$. Nous pouvons remarquer que, bien que ces nouveaux arcs, que nous appelons arcs d'horloge, ne doivent pas avoir d'effet sur le coût du *retiming* (puisqu'il ne font pas partie du graphe original, nous ne souhaitons pas les compter en tant qu'arcs de poids nul), ils peuvent avoir une influence sur le coût du flot puisqu'ils contraignent les changements de poids. En considérant ce nouveau problème, nous pouvons mieux comprendre notre algorithme : le flot correspond bien aux contraintes structurelles du graphe, et son coût agit bien comme une limite inférieure au coût du *retiming*.

Changements dans l'algorithme

Nous montrons ici comment incorporer ces nouveaux arcs de manière harmonieuse à notre algorithme et comment leur donner une coloration compatible avec les couleurs définies précédemment.

Le problème est le suivant : étant donné un graphe de dépendance $G = (V, E, d)$, nous souhaitons trouver un *retiming* légal $r : V \to \mathbb{Z}$ de $G' = (V, E', d')$ où :

$$d'(e) = \begin{cases} d(e) & \text{si } e \in E \\ W(u,v) - 1 & \text{si } e \in E_{\text{clock}} \end{cases}$$

et bien sûr qui minimise le nombre d'arcs de poids nul dans E seulement.

Pour cela, nous allons garder la même définition de v_r et $z_{f,r}(e)$ pour les arcs $e \in E$. Pour les arcs $e \in E_{\text{clock}}$, nous définissons $v_r(e) = 0$ et $z_{f,r}(e) = -f(e)d_r(e)$. Ceci nous garantit toujours que $v_r(e) \geq z_{f,r}(e)$ pour tous les arcs, incluant les nouveaux arcs d'horloge, donc les indices de conformité sont toujours positifs. Les coûts $V(r)$ du *retiming* et $Z(f)$ du flot sont toujours définis de la même manière, mais comme somme sur tous les arcs (incluant les arcs d'horloge) : $V(r) = \sum_{e \in E'} v_r(e)$ et $Z(f) = \sum_{e \in E'} z_{f,r}(e)$. La proposition 2 reste vraie pour son équivalent avec arcs d'horloge, ce que nous montrons dans la proposition suivante :

Proposition 5 *Soit f un flot de G', C_f l'ensemble des arc e tels que $f(e) > 0$ et $|C_f|$ le nombre d'arcs dans C_f. Alors pour tout* retiming *légal r de G,*

$$\sum_{e \in E'} z_{f,r}(e) = \sum_{e \in E'} z_{f,0}(e).$$

Preuve

$$\begin{aligned}
\sum_{e \in E'} z_{f,r}(e) &= \sum_{e \in C_f \cap E} (1 - f(e)d_r(e)) + \sum_{e \in C_f \cap E_{\text{clock}}} (-f(e)d_r(e)) \\
&= |C_f \cap E| - \sum_{e=(u,v) \in E'} f(e)(d(e) + r(v) - r(u)) \\
&= |C_f \cap E| - \sum_{e \in E'} f(e)d(e) - \sum_{e=(u,v) \in E'} f(e)r(v) + \sum_{e=(u,v) \in E'} f(e)r(u) \\
&= |C_f \cap E| - \sum_{e \in E'} f(e)d(e) - \sum_{v \in V} r(v) \underbrace{\left(\sum_{e=(.,v) \in E'} f(e) - \sum_{e=(v,.) \in E'} f(e) \right)}_{= 0 \text{ puisque } f \text{ est un flot}} \\
&= |C_f \cap E| - \sum_{e \in E'} f(e)d(e) \\
&= \sum_{e \in C_f \cap E} (1 - f(e)d(e)) + \sum_{e \in C_f \cap E_{\text{clock}}} (-f(e)d(e)) \\
&= \sum_{e \in E'} z_{f,0}(e), \text{le coût de } f \text{ pour le } \textit{retiming} \text{ nul}
\end{aligned}$$

□

C'est donc sans grande surprise que l'on découvre que malgré les arcs d'horloge, le coût d'un flot pour un *retiming* donné r est égal à son coût pour le *retiming* nul. Le coût total d'un flot ne dépend toujours pas du *retiming* du graphe.

Quant aux propositions 3 et 4, elles peuvent rester exactement telles quelles, la preuve est toujours valide. Nous sommes alors en droit de nous demander si les choses ont vraiment changé. En fait, les choses ont effectivement changé. Les mécanismes de la minimisation du nombre d'arcs

de poids nul sont effectivement toujours les mêmes (excepté que cette fois nous ne comptons pas les arcs d'horloge), mais en raison des arcs d'horloge, les *retimings* autorisés ne sont plus les mêmes. Nous devons rester conscients du fait que nous travaillons sur un graphe plus gros dont certains arcs sont de nouvelles contraintes. À cause de ces nouveaux arcs, tout *retiming* légal r de G' possède une propriété très intéressante : $\Phi(G'_r) \leq \Phi$, en d'autres termes la période d'horloge du résultat ne dépasse pas la constante que nous nous somme fixés initialement. Évidemment, la constante choisie doit être une période d'horloge réalisable, autrement il n'existera pas de *retiming* légal pour G' (à cause du théorème 1).

Il nous reste encore à déterminer la couleur des nouveaux arcs à l'aide d'un nouveau diagramme de conformité spécifique aux arcs d'horloge.

- si $f(e) = 0$ et $d_r(e) = 0$, l'arc sera noir ;
- si $f(e) = 0$ et $d_r(e) > 0$, l'arc sera incolore ;
- si $f(e) > 0$ et $d_r(e) = 0$, l'arc sera rouge ;
- dans tous les autres cas, l'arc sera vert.

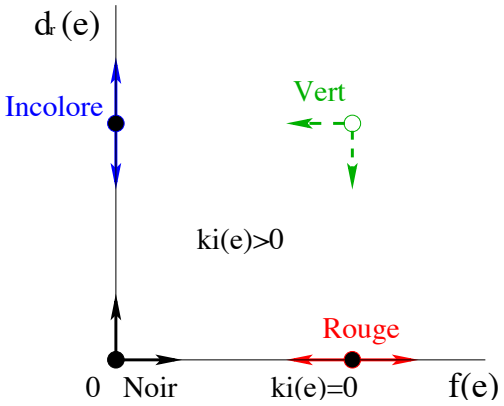


FIG. 2.11 – Diagramme de conformité et couleur des arcs d'horloge.

Note : nous pouvons remarquer que nous devons partir d'un graphe légal, c'est-à-dire tel que le poids de tous les arcs soit positif ou nul (ce qui n'est pas le cas des arcs d'horloge si nous partons d'un graphe quelconque). Un tel graphe est obtenu en appliquant l'algorithme de minimisation de la période d'horloge au graphe initial. À partir de ce point, si nous partons également du flot nul, tous les arcs d'horloge sont conformes, et nous ne créerons jamais d'arcs d'horloge non conformes.

Complexité

L'algorithme final est le même que l'algorithme 10 (excepté pour l'étape de coloration des arcs qui doit colorier différemment les arcs du graphe original et les arcs d'horloge). Il trouve un *retiming* optimal, dont la légalité garantit au résultat une période d'horloge d'au plus Φ. Le nombre d'étapes est toujours le même puisque tous les arcs d'horloge sont conformes et le restent, mais le nombre total d'arcs a augmenté puisque G' possède au pire $|V|^2$ arcs d'horloge. Ainsi, la procédure de marquage est maintenant en $O(|E| + |V|^2)$ et l'algorithme total en $O(|E|^2 + |E||V|^2)$.

Note : pour la réalisation pratique, il est généralement possible d'éliminer un certain nombre d'arcs redondants. En effet, en tirant parti à la fois du théorème 1 et du lemme 1, nous pouvons remarquer que si plusieurs arcs se trouvent entre un même couple de sommets, il n'est utile de

garder que ceux ayant un poids minimum, car les autres n'auront jamais un poids nul tant que le *retiming* restera légal. Dans le même esprit, un arc d'horloge entre un couple de sommets (u, v) est redondant s'il existe déjà un arc de poids inférieur ou égal à $W(u, v)$ entre ces deux sommets. Voir [70] pour plus de détails.

2.5 Résultats expérimentaux

Nous avons mené un certain nombre d'expériences d'une part afin d'évaluer à la fois l'efficacité absolue de notre technique et l'apport de la minimisation du nombre d'arcs de poids nul dans l'heuristique CDR initiale, et d'autre part afin de comparer notre technique avec une autre approche décomposée du pipeline logiciel. Ceci ayant pour but de déterminer comment envisager le pipeline logiciel décomposé dans son ensemble. Comme le fait remarquer B. Rau dans [93], une grande partie des boucles dans les *programmes étalons* (*benchmarks*) classiques sont si simples qu'elles peuvent être ordonnancées de façon optimale avec la plupart des heuristiques, même les moins élaborées. Cette constatation nous permet de conclure deux choses : premièrement, les *programmes étalons* sont *a priori* représentatifs des "vrais" programmes[7], mais pas nécessairement des cas difficiles. Deuxièmement, il devient clair que la complexité d'une heuristique d'ordonnancement est un facteur déterminant dans son intérêt : inutile de recourir à une recherche exponentielle d'une solution optimale si un simple algorithme glouton est à même de trouver cette solution dans 99% des cas. Notre heuristique reste intéressante car elle est polynomiale (avec un polynôme de faible degré), mais elle ne présente aucune valeur si d'autres heuristiques plus simples sont aussi performantes. Nous espérons donc montrer que nous obtenons de meilleurs résultats que les autres méthodes de façon générale, mais surtout lorsque les cas sont difficiles (puisque c'est là que va se faire la différence).

Nous avons développé dans ce but une plate-forme d'expérimentation permettant d'effectuer des statistiques de performance sur des jeux de graphes générés aléatoirement. Nous avons appelé cette plate-forme PASTAGA (pour Plateforme d'Analyse Statistique et de Tests d'Algorithmes sur Graphes Aléatoires). Différents paramètres peuvent être fournis au générateur de graphes aléatoires contenu dans PASTAGA afin d'assurer le réalisme des cas à résoudre. Bien sûr, la raison initiale de ce choix est fondée d'une part sur la difficulté à générer automatiquement un graphe de dépendance à partir du code en langage machine d'un programme[8], d'autre part sur la difficulté à trouver des programmes représentatifs de l'ensemble des situations possibles (et néanmoins réalistes). Finalement, nous avons trouvé l'approche particulièrement bénéfique : la variété des exemples produits par le générateur aléatoire nous a fortement aidés lors des phases de déboguage, nous a permis d'identifier les cas difficiles, de mieux comprendre les interactions entre *retiming* et ordonnancement, et de déterminer quels étaient les points forts des différentes approches.

De façon détaillée, notre suite logicielle a les caractéristique suivantes :

Description de l'architecture

- la description de l'architecture cible est donnée à l'aide d'un vecteur indiquant la quantité disponible pour chaque ressource (unités fonctionnelles, mais cela peut inclure aussi les registres ou le bus mémoire, etc.) ;
- chaque opération autorisée dans l'architecture cible est décrite à l'aide d'une table de ré-

[7] Évidemment, la représentativité des *programmes étalons* en général est également discutable. La plupart d'entre eux portent sur des calculs scientifiques, et sont souvent optimisés dans un souci de réutilisation maximale de la mémoire, ce qui va à l'encontre du parallélisme.

[8] Notre algorithme est destiné à être intégré dans une chaîne de compilation complète avec une analyse de dépendance de relativement haut niveau, ce qui n'est pas disponible à l'heure actuelle.

servation des ressources et d'un temps d'exécution[9]. À l'aide de ce modèle, il est possible de simuler tous types de situations, unités homogènes ou non, pipelinées ou non et unités fonctionnelles plus complexes ;
- les informations sont décrites dans un fichier faisant partie des données d'entrée du simulateur ;
- la proportion des différents types d'opérations dans les graphes générés est un paramètre.

Structure des graphes
- les graphes sont générés aléatoirement de telle sorte qu'ils ne comportent aucun circuit de poids nul ;
- les poids sont tirés aléatoirement de manière à respecter une proportion donnée en paramètre (par exemple 67% de poids nuls, 33% de 1) ;
- le nombre de sommets et d'arcs du graphe sont des paramètres, ainsi que le degré entrant maximum d'un sommet (par exemple 2 en pratique pour les opérations arithmétiques simples) ;
- il est possible de filtrer les graphes en fonction de leur nombre de composantes fortement connexes afin de mettre en évidence les différents cas de figure.

Statistiques sur l'*intervalle de lancement*
- la borne inférieure due aux contraintes de dépendance (RecMII = λ_∞) est calculée, ainsi qu'un ordonnancement optimal σ_∞ sans contraintes de ressources ;
- la borne inférieure due aux ressources (ResMII) est calculée (remarquons toutefois que cette borne n'est pas nécessairement atteinte, même pour un ensemble d'opérations indépendantes dans le cas de modèles de ressources complexes) ;
- il est possible de filtrer ou de compter les graphes pour lesquels trop peu de ressources sont disponibles pour réaliser l'ordonnancement optimal σ_∞ (ces cas sont intéressants car le problème devient plus difficile).

Statistiques sur l'ordonnancement
- le code résultant d'une boucle après son réordonnancement comporte en général un prologue et un épilogue (servant respectivement à amorcer et à vider le pipeline). Sa taille est évaluée. Elle dépend de Δr, la différence maximale entre deux valeurs de *retiming*, et de la taille du corps de la boucle ;
- les besoins en registres de l'ordonnancement sont évalués en calculant *maxlive*, le nombre maximum de valeurs vivantes à un instant donné de l'ordonnancement, c'est-à-dire calculées mais pas encore utilisées.

Heuristiques mises en œuvre
- nous mettons en œuvre diverses stratégies d'ordonnancement cyclique fondées sur la compaction de boucle : compaction seule, compaction après minimisation de la période d'horloge (avec l'algorithme de Leiserson et Saxe, voir section 1.2.3), compaction après minimisation du nombre d'arcs de poids nul (voir section 2.4.2), compaction après minimisation du nombre d'arcs de poids nul en conservant une période d'horloge minimale (voir la section 2.4.3) et compaction à partir de l'ordonnancement en ressources illimitées (approche de Gasperoni et Schwiegelshohn, voir [48]) ;

[9]Cet ensemble de tests a été réalisé avant que nous n'utilisions le modèle plus général incluant les latences sur les arcs du graphe et non sur les sommets. Il correspond au cas où tous les arcs sortant d'un sommet ont la même latence. Néanmoins, nous ne pensons pas que cela puisse changer nos conclusions. En effet nos remarques portent plus sur la structure des graphes et les heuristiques globales d'ordonnancement, indépendamment du détail de compaction sur chaque arc. De plus, ce modèle est le modèle employé dans la plupart des heuristiques de pipeline logiciel décomposé.

- différentes stratégies d'ordonnancement sont disponibles : au plus tôt, au plus tard, sélection du meilleur des deux, etc. combinées à l'ajout de diverses heuristiques sur le choix des opérations ;
- nous avons ajouté la possibilité de réduire l'*intervalle de lancement* obtenu en recouvrant les motifs successifs dans la limite autorisée par les contraintes de ressources et de dépendance (voir [97]). Cet ajout est important dans le cas de ressources pipelinées afin d'atteindre l'optimal. Le recouvrement se fait en réduisant l'*intervalle de lancement* tant qu'aucune contrainte n'est violée. Les contraintes de dépendances sont vérifiées en utilisant l'inégalité 2.6 pour $d_r(e) > 0$ (les contraintes correspondant aux arcs de poids nul sont satisfaites par les dates choisies par la compaction), autrement dit :

$$\lambda \geq \frac{l(e) + a'_u - a'_v}{d(e) + r(v) - r(u)} \tag{2.11}$$

  tandis que les contraintes de ressources sont vérifiées à partir des dates déterminées par la compaction et des tables de réservation des ressources, en recouvrant plusieurs motifs successifs lorsqu'une opération poursuit son exécution sur plusieurs itérations. Nous proposons plusieurs méthodes de recouvrement :
  - un recouvrement limité abaissant λ tant que les dépendances sont respectées et que toutes les dates déterminées par la compaction restent strictement inférieures à λ. Autrement dit, nous nous limitons à un recouvrement ne créant pas de *retiming* artificiel (voir la section 2.3.2 sur la décomposition d'un ordonnancement cyclique en un *retiming* et une compaction) ;
  - un recouvrement maximal abaissant λ jusqu'à la limite des contraintes, pouvant résulter en un *intervalle de lancement* plus petit que les dates d'ordonnancement des opérations. Dans ce cas, nous pouvons appliquer la décomposition proposée à la section 2.3.2 pour trouver un nouveau *retiming* : le recouvrement crée alors un *retiming* artificiel (nous sortons légèrement du modèle « *retiming* déterminé une fois pour toutes, puis compaction ») ;
  - un recouvrement maximal couplé à une modification du motif améliorant son efficacité (nous donnerons plus de détails lors de l'interprétation des mesures).

Nous donnons maintenant nos résultats sur différents modèles de machines, premièrement une seule unité fonctionnelle pipelinée (comme dans notre exemple principal présenté en section 2.3.1), puis une machine VLIW plus complexe. Nous avons également effectué de nombreux tests sur machines non pipelinées, mais ce cas semble moins réaliste et nous ne l'aborderons que rapidement en section 2.5.3 (en fait, notre heuristique est encore meilleure sur de telles configurations, mais le cas ne se rencontre pas vraiment en pratique). Pour l'ensemble de ces mesures, nous avons choisi de générer des graphes possédant 67% de dépendances indépendantes de la boucle (nulles) et 33% de dépendances portées par la boucle (de poids 1). Une telle configuration est un cas difficile pour le pipeline logiciel, en effet un nombre important de dépendances indépendantes de la boucle rend l'ordonnancement très contraint, en particulier pour les graphes fortement connexes. Néanmoins, nous estimons ces valeurs plus réalistes et représentatives d'un calcul effectué majoritairement à l'intérieur de la boucle et pour lequel un petit nombre de valeurs (résultats finaux) sont réutilisées dans l'itération suivante. Le degré entrant maximum d'une opération est fixé à 2 et le degré entrant moyen à $1,8$ (toutes les opérations n'ont pas deux opérandes, mais parfois seulement un ou aucun). Les abréviations à l'intérieur des tables ont la signification suivante : Comp pour compaction seule, Clock pour compaction après minimisation de la période d'horloge (c'est-à-dire l'algorithme de Calland, Darte et Robert), MinEdge pour compaction après minimisation du nombre d'arcs de poids

nul, ClockEdge pour compaction après minimisation du nombre d'arcs de poids nul en conservant une période d'horloge minimale, et GS pour l'heuristique de Gasperoni et Schwiegelshohn. *Maxlive* est le nombre maximal de variables en vie à un instant donné de l'ordonnancement (en supposant que tous les arcs du graphe correspondent à des dépendances de flot). Δr est la différence maximale entre la valeur de *retiming* de deux instructions (et correspond à la taille du prologue et de l'épilogue du code généré). Le pourcentage de cas atteignant la borne inférieure ainsi que la distance moyenne et maximum à cette borne sont indicatifs de la qualité absolue de l'heuristique d'ordonnancement. En effet, les tests atteignant cette borne sont bien sûr optimaux, mais de façon générale la borne n'est pas nécessairement atteignable. Ainsi le pourcentage de tests atteignant la borne inférieure est lui-même inférieur au pourcentage de tests optimaux. Toutes les valeurs données sont les résultats de mesures sur un échantillon de 1000 graphes aléatoires. Les temps moyens d'exécution de chaque phase de l'approche décomposée pour un graphe sont donnés, la machine utilisée pour ces mesures est un PC sous Linux, ayant un processeur Duron d'AMD cadencé à 700MHz sur carte mère Soltek 75KAV avec 384 méga-octets de SDRAM. Pour ClockEdge, le temps indiqué pour la phase de *retiming*, n'inclus pas le calcul de la période d'horloge minimale (qui est calculée par Clock). Pour GS, le calcul de l'ordonnancement en ressources infinies et du *retiming* associé est inclus dans le temps d'exécution de la phase de compaction.

2.5.1 Une seule unité fonctionnelle pipelinée

Dans cette section, nous utilisons une machine possédant une seule unité fonctionnelle pipelinée ayant plusieurs délais différents pour ses opérations. Notre modèle est très similaire au microprocesseur LANai 3.0 de Myricom [81]. Ce microprocesseur offre un modèle très simple, mais le cas d'une unique unité fonctionnelle pipelinée est illustratif d'une partie des processeurs répandus dans le grand public (évidemment, depuis le MMX d'Intel, les processeurs grand public sont maintenant partiellement SIMD, du moins en théorie). Nous nous proposons d'étudier les effets du changement de plusieurs paramètres sur les différentes heuristiques.

La table 2.1 réunit les résultats pour deux types d'opérations différentes, de délais respectifs 3 et 1 (ce sont les délais réels de la nouvelle version du processeur LANai, 3 pour un *load* et 1 pour les autres opérations). Les graphes ont 10 sommets donc 18 arcs (à cause du degré entrant à $1,8$), dont 12 dépendances indépendantes de la boucle. Ces statistiques montrent que minimiser le nombre de

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	12.9	11.4	11.4	11.2	11.5
Tests meilleurs que la Comp	100%	99.7%	95.3%	99.3%	99.8%
Tests strictement meilleurs que la Comp	0%	65.3%	66.5%	72%	59.4%
Gain moyen sur la Comp	0	1.48	1.46	1.65	1.37
Tests atteignant la borne inférieure	23.1%	72.7%	76%	86.2%	65.5%
Distance moyenne à la borne inférieure	1.81	0.334	0.355	0.161	0.443
Distance maximale à la borne inférieure	9	3	7	3	3
Valeur moyenne de *maxlive*	6.08	7.08	9.74	9.78	6.89
Valeur moyenne de Δr	0	1.04	2.54	2.52	0.858
Temps d'exécution (*retiming*)	0ms	1.03ms	0.31ms	0.67ms	0ms
Temps d'exécution (scheduling)	1.15ms	1.42ms	1.43ms	2.58ms	1.69ms

TAB. 2.1 – Résultats pour une unité fonctionnelle pipelinée et 10 opérations.

contraintes pour la compaction est une approche payante (MinEdge et ClockEdge sont les deux

heuristiques atteignant le plus souvent la borne inférieure). En outre, il apparaît clairement que la minimisation de la longueur du chemin critique pour la compaction permet d'être en moyenne plus proche de l'optimal (comme l'indique la distance moyenne à la borne de Clock). Il est donc logique que les deux approches combinées en ClockEdge produisent le meilleur résultat : l'heuristique bénéficie des deux optimisations et s'avère plus robuste. Les différences sur les deuxième et troisième lignes montrent bien la séparation entre les différentes approches :

- Clock et GS ne changent que peu le graphe (Δr est très bas) et par conséquent sont généralement proches de la compaction, lorsqu'elles ne sont pas strictement meilleures elles sont aussi bonnes car elles compactent un graphe peu modifié ;
- MinEdge et ClockEdge changent plus fortement le graphe (fortes valeurs de Δr), la conséquence est qu'elles travaillent sur un graphe profondément modifié par rapport à celui utilisé par la compaction seule. Les cas sur lesquels ces heuristiques échouent sont donc différents, d'où le pourcentage légèrement moins élevé de cas aussi bons que la compaction seule.

Une question semble légitime vis-à-vis de l'heuristique de minimisation du nombre d'arcs de poids nul : y a-t-il suffisamment d'opportunités d'optimisation dans un graphe fortement connexe pour que la minimisation du nombre d'arcs de poids nul ait le moindre effet ? Peut-être le bénéfice apparent de notre algorithme n'est-il dû qu'au fait que des contraintes disparaissent entre différentes composantes fortement connexes du graphe ? La réponse est non : la minimisation du nombre d'arcs de poids nul est bel et bien efficace sur l'ensemble du graphe. La table 2.2 recense les performances des diverses heuristiques lorsque l'échantillon de 1000 graphes n'est composé que de graphes fortement connexes. Ces résultats montrent que des conclusions similaires peuvent être données dans le cas de

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	14.1	13.1	13.6	13.1	13.2
Tests meilleurs que la Comp	100%	98.8%	77.9%	92.5%	99.3%
Tests strictement meilleurs que la Comp	0%	43.2%	40.7%	51.8%	38%
Gain moyen sur la Comp	0	0.948	0.493	1.03	0.855
Tests atteignant la borne inférieure	28.7%	49.7%	37.8%	55%	44.7%
Distance moyenne à la borne inférieure	1.61	0.659	1.11	0.574	0.752
Distance maximale à la borne inférieure	9	3	7	4	4
Valeur moyenne de *maxlive*	5.79	5.89	6.11	6	5.86
Valeur moyenne de Δr	0	0.623	1.37	1.27	0.485
Temps d'exécution (*retiming*)	0ms	1.25ms	0.15ms	0.8ms	0ms
Temps d'exécution (scheduling)	1.17ms	1.16ms	1.34ms	2.37ms	1.49ms

TAB. 2.2 – Graphes fortement connexes seulement.

graphes fortement connexes, avec un effet légèrement plus faible dû au fait que la forte connexité limite tout de même les possibilités d'amélioration de l'heuristique par rapport aux autres solutions. Le fait que moins d'exemples atteignent la borne inférieure ne signifie pas nécessairement qu'il y a moins de cas optimaux. Dans le cas de graphes fortement connexes, le problème est en effet plus complexe, et l'interférence entre contraintes de ressources et contraintes de dépendance peut rendre plus fréquemment la borne impossible à atteindre par l'optimal.

Il est également intéressant de remarquer sur la table 2.2 que, comme nous pouvions nous en douter, la valeur moyenne de *maxlive* est bien plus faible dans le cas de graphe fortement connexe (le fait que le *retiming* ne change pas le poids d'un circuit laissait augurer ce phénomène). Ceci était faux dans la table précédente où *maxlive* variait de 6.08 à 9.79 selon l'heuristique employée.

Une possibilité pour faire baisser cette valeur dans les heuristiques à base de *retiming* est d'essayer, une fois l'ordonnancement déterminé, de changer à nouveau les poids des arcs par un *retiming* additionnel dans le but d'atteindre la plus petite valeur possible pour *maxlive*. Cette technique est connue sous le nom de réordonnancement par étages (*stage scheduling*) et a été introduite par Eichenberger et al. dans [34]. Elle est souvent utilisée dans le contexte du *modulo scheduling*. Pour ces tests, nous n'avons réalisé qu'une approche plus naïve fondée, une fois encore, sur le *retiming* : nous n'en donnons pas les détails, mais l'idée générale est d'essayer de ramener les poids à une valeur la plus proche possible de leur valeur d'origine une fois l'ordonnancement effectué. Ceci est réalisé grâce à une variante de l'algorithme 2 (page 22) et à une légère transformation du graphe. À l'aide de cette approche, *maxlive* descend de 9.79 à 7.73. Nous présentons d'un point de vue théorique la technique complète du réordonnancement par étages au chapitre 3.

Nous indiquons en table 2.3 l'évolution des heuristiques lorsque le nombre de sommets augmente. Sans grande surprise, nous constatons que les performances augmentent avec le nombre de sommets : plus de sommets, même degré entrant moyen, donc moins d'arcs, des opérations de petit délai, donc plus de facilité à remplir les ressources. Des expérimentations avec encore plus de sommets ont

	Comp	Clock	MinEdge	ClockEdge	GS
10 sommets					
Intervalle de lancement moyen	12.9	11.4	11.4	11.2	11.5
Tests atteignant la borne inférieure	23.1%	72.7%	76%	86.2%	65.5%
Distance moyenne à la borne inférieure	1.81	0.334	0.355	0.161	0.443
15 sommets					
Intervalle de lancement moyen	16.9	15.7	15.8	15.6	15.7
Tests atteignant la borne inférieure	44.1%	82.4%	83.5%	91%	78.8%
Distance moyenne à la borne inférieure	1.41	0.224	0.299	0.119	0.294
20 sommets					
Intervalle de lancement moyen	21	20.2	20.3	20.2	20.3
Tests atteignant la borne inférieure	66.9%	93.1%	91.9%	96.6%	91.4%
Distance moyenne à la borne inférieure	0.869	0.094	0.169	0.037	0.124

TAB. 2.3 – Variation du nombre de sommets.

montré la même évolution dans toute la suite de nos mesures. C'est pourquoi nous nous limitons à un nombre de sommets aussi petit : nous souhaitons avant tout identifier les cas difficiles et les faiblesses des différentes approches.

2.5.2 Plusieurs unités fonctionnelles pipelinées

Nous considérons maintenant un modèle de machines pour lequel 4 opérations peuvent être démarrées simultanément à chaque instant (4 unités complètement pipelinées). Nous disposons de 5 types d'opérations différents de délais respectifs 1, 2, 3, 5, et 20, avec une répartition de 30% d'opérations de délais 1 et 2 (opérations simples et fréquentes, comme par exemple des additions, décalages, ...), 15% de délais 3 et 5 (opérations plus coûteuses mais néanmoins nombreuses, comme par exemple des multiplications, divisions, fonctions élémentaires, ...), 10% de délai 20 (opérations très coûteuses, comme par exemple un accès mémoire). Cette situation devrait être relativement simple à résoudre puisque les ressources abondent (en particulier avec des unités fonctionnelles pipelinées). La table 2.4 présente les résultats obtenus en appliquant notre heuristique exactement telle que décrite en section 2.4.3 et l'heuristique GS exactement telle que décrite dans [48]. Ces

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	29.2	25.3	26.1	24.9	25.1
Tests meilleurs que la Comp	100%	99.1%	85.2%	98.4%	99.9%
Tests strictement meilleurs que la Comp	0%	66.6%	62.9%	73.1%	68.1%
Gain moyen sur la Comp	0	3.9	3.02	4.27	4.06
Tests atteignant la borne inférieure	24.6%	72.1%	52%	76.7%	57.2%
Distance moyenne à la borne inférieure	5.7	1.8	2.68	1.44	1.64
Distance maximale à la borne inférieure	46	17	30	17	19
Valeur moyenne de *maxlive*	5.81	6.13	8.56	8.62	6.18
Valeur moyenne de Δr	0	0.976	2.54	2.56	0.953
Temps d'exécution (*retiming*)	0ms	1.29ms	0.32ms	0.77ms	0ms
Temps d'exécution (scheduling)	1.52ms	1.46ms	1.41ms	2.98ms	1.81ms

TAB. 2.4 – Résultats sur une machine VLIW pipelinée à 4 unités fonctionnelles.

résultats s'avèrent plutôt faibles compte tenu des ressources disponibles et laissent penser que nous perdons un certain nombre d'opportunités d'optimisation. La première remarque que l'on peut faire concerne le recouvrement des motifs. Jusqu'alors, l'*intervalle de lancement* était déterminé en recherchant le premier instant après le démarrage de la dernière opération à partir duquel il était possible de commencer une nouvelle itération (ce que nous avons présenté sous la dénomination de recouvrement limité dans la description de notre outil). En fait, il est possible de chercher un recouvrement plus tôt à l'intérieur de notre ordonnancement acyclique, mais dans ce cas le motif de l'ordonnancement et le *retiming* changent. C'est ce que nous avons présenté sous la dénomination de recouvrement maximal dans la description de notre outil. Dans l'exemple précédent, cette approche n'était pas requise car les ressources étaient vite saturées et peu d'améliorations étaient possibles par recouvrement (nous avons pu le vérifier expérimentalement). Nous avons donc ajouté ce recouvrement maximal à nos heuristiques en cherchant, une fois l'ordonnancement réalisé, le plus petit *intervalle de lancement* satisfaisant les contraintes de ressources et de dépendance. Ceci nous a effectivement permis d'améliorer la compaction de boucle (voir table 2.5). En revanche cette approche n'a quasiment aucun effet sur les autres heuristiques qui, d'une certaine manière, font déjà le travail en déterminant leur *retiming* dans le but de recouvrir les itérations initiales de la boucle.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	26.5	25.2	26.1	24.9	25.1
Tests meilleurs que la Comp	100%	91%	75.3%	92.2%	86.7%
Tests strictement meilleurs que la Comp	0%	35.9%	33.4%	40.8%	32.7%
Gain moyen sur la Comp	0	1.31	0.42	1.59	1.37
Tests atteignant la borne inférieure	53.9%	74.1%	54.6%	76.9%	57.2%
Distance moyenne à la borne inférieure	3.02	1.7	2.6	1.42	1.64
Distance maximale à la borne inférieure	27	17	30	17	19
Valeur moyenne de *maxlive*	5.95	6.12	8.55	8.62	6.18
Valeur moyenne de Δr	0.593	0.98	2.54	2.56	0.953
Temps d'exécution (*retiming*)	0ms	1.39ms	0.24ms	0.82ms	0ms
Temps d'exécution (scheduling)	1.52ms	1.45ms	1.55ms	2.81ms	1.95ms

TAB. 2.5 – Avec un recouvrement maximal (impliquant un changement du *retiming*).

Ainsi, la compaction de boucle peut être « sauvée » grâce à de plus gros recouvrements. Cependant, cela n'explique toujours pas le fait qu'avec autant de ressources disponibles l'ensemble des heuristiques se comporte aussi mal (ou plutôt « aussi peu bien »). En examinant de plus près les mécanismes sous-jacents à notre approche du pipeline logiciel décomposé, il apparaît que l'ordonnancement au plus tôt a tendance à remplir toutes les ressources disponibles au tout début du motif. C'est ce phénomène qui limite fortement les possibilités de recouvrement : impossible de rapprocher deux itérations si la deuxième a besoin de toutes les ressources dès son commencement. Calland, Darte et Robert avaient déjà pressenti ce problème dans leur article [10], mais leur solution, consistant à ordonnancer alternativement au plus tôt puis au plus tard les opérations, n'est pas satisfaisante lorsque plus de deux motifs doivent se recouvrir. Nous avons donc mis en place une heuristique à la fois plus simple et plus efficace en pratique : lorsque nous souhaitons recouvrir les motifs successifs et que nous sommes bloqués par des conflits de ressources, nous essayons de repousser vers la fin de l'ordonnancement toutes les opérations ayant une liberté de mouvement (sans changer l'ordre des opérations, nous ne faisons pas vraiment de réordonnancement). Nous répétons l'opération jusqu'à ce qu'il ne soit plus possible de pousser ou de recouvrir plus. Cette heuristique supplémentaire bien que simple donne d'excellents résultats comme nous pouvons le constater sur la table 2.6.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	24.4	24.2	24.1	23.9	24
Tests meilleurs que la Comp	100%	90.8%	90%	92.7%	88.1%
Tests strictement meilleurs que la Comp	0%	14.1%	17.1%	18.2%	15.2%
Gain moyen sur la Comp	0	0.184	0.327	0.442	0.394
Tests atteignant la borne inférieure	78.3%	82.9%	83.1%	87.2%	76.9%
Distance moyenne à la borne inférieure	0.92	0.736	0.593	0.478	0.526
Distance maximale à la borne inférieure	25	15	15	15	10
Valeur moyenne de *maxlive*	6.02	6.04	8.44	8.52	6.21
Valeur moyenne de Δr	0.808	1.01	2.51	2.55	0.965
Temps d'exécution (*retiming*)	0ms	1.18ms	0.24ms	0.56ms	0ms
Temps d'exécution (scheduling)	6.72ms	3.59ms	5.56ms	9.82ms	4.45ms

TAB. 2.6 – Avec modification de l'ordonnancement lors du recouvrement.

L'ajout de cette heuristique met bien en évidence le problème lié à l'emploi d'un ordonnancement au plus tôt et l'importance de la politique d'ordonnancement acyclique : d'une manière ou d'une autre, l'heuristique d'ordonnancement acyclique utilisée pour la compaction doit être consciente de la nature cyclique du motif qu'elle produit, il est important de faire en sorte que les motifs successifs puissent s'emboîter au mieux. Cette dernière remarque est plus importante qu'il n'y paraît et notre heuristique poussant les opérations n'est pas l'unique optimisation qui nous a permis d'obtenir de meilleures performances. Comme le lecteur pourra s'en douter, il n'y a aucune raison pour que les contraintes de ressources soient les seules à poser problème lors du recouvrement : les contraintes de dépendance entre les dernières opérations ordonnancées et les premières du motif suivant peuvent poser problème. Ceci est résolu dans notre fonction de priorité pour le choix des opérations à ordonnancer : comme dans un ordonnancement au plus tôt classique avec contraintes de ressources, nous choisissons d'abord les opérations ayant le plus long chemin de dépendance vers un puits du graphe. Connaissant cette priorité, nous savons quelles opérations seront ordonnancées en premier dans notre graphe. Ainsi, en cas d'égalité, nous choisissons en priorité l'opération dont

les successeurs dans le motif suivant seront ordonnancés d'abord (afin qu'elle ne se retrouve pas à la fin du motif). En résumé, même si le problème d'ordonnancement est acyclique, l'algorithme utilisé doit faire un choix intelligent des opérations. Une fois l'ordonnancement réalisé, notre heuristique de poussée semble alors tout à fait efficace dans la plupart des cas. La figure 2.12 représente deux

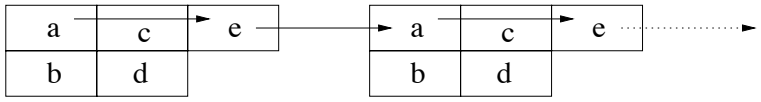


FIG. 2.12 – Un exemple de mauvais ordonnancement acyclique.

motifs successifs d'un ordonnancement cyclique sur deux ressources, ainsi que les dépendances entre opérations (toutes de temps de réservation et de latence égales à 2). Sur cet exemple, les opérations ont été ordonnancées au plus tôt sans priorité particulière. Les motifs ne peuvent pas se recouvrir (à la fois à cause de la dépendance de e vers a dans le motif suivant et des ressources saturées au début du motif), l'*intervalle de lancement* est de 6. En utilisant notre heuristique de poussée et

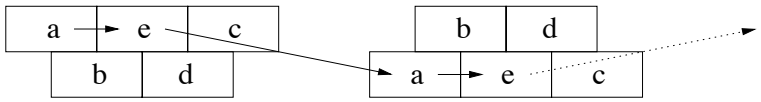


FIG. 2.13 – L'exemple 2.12 corrigé par nos améliorations.

notre fonction de priorité pour réordonnancer le motif, nous obtenons le résultat représenté sur la figure 2.13 ; cette fois le recouvrement est possible et l'*intervalle de lancement* est de 5. Notons que, dans notre solution, l'allocation des ressources n'est pas cyclique. Si l'assignation des opérations aux ressources doit se faire de manière explicite, il est possible de dérouler la boucle pour retrouver une allocation cyclique.

La toujours « mauvaise » performance de GS reste néanmoins difficile à expliquer. En effet, la force de cette heuristique est qu'elle part de l'ordonnancement optimal en ressources infinies et essaie de l'« écraser » pour respecter les contraintes de ressources réelles. Dans le cas qui nous intéresse, puisque nous disposons de beaucoup de ressources, nous pouvons attendre un résultat meilleur. Dans [48] cette heuristique n'est pas présentée en termes de *retiming*, mais nous pouvons la réinterpréter. L'heuristique utilise les dates produites par l'ordonnancement en ressources infinies σ_∞. En utilisant la même technique qu'à la section 2.3.2, nous pouvons décomposer cet ordonnancement en une combinaison de *retiming* et de compaction. Cette compaction fournit pour chaque opération une date de départ dans le motif. Le problème lors d'un recouvrement est qu'une opération ordonnancée à la fin du motif peut avoir des successeurs qui dépendent de lui ordonnancés au début du motif suivant. L'idée de Gasperoni et Schwiegelshohn est, pour toute opération dont l'exécution chevauche deux motifs successifs, d'imposer à tous ses successeurs une date de début minimale dans le motif : celle calculée à partir de l'ordonnancement en ressources infinies. Cette idée leur permet d'éliminer le terme $\max_{u \in V} t_\Delta(u)$ de la borne lorsque les ressources sont pipelinée. Nous souhaitons aller plus loin : ces dates de début minimales choisies par Gasperoni et Schwiegelshohn ont un intérêt théorique, mais elles n'empèchent pas le motif de présenter les défauts que nous avons décrits et qui interdisent un bon recouvrement. Notre idée est qu'en restant le plus proche possible de l'ordonnancement en ressources infinies, l'heuristique GS peut espérer atteindre la même performance lorsque suffisamment de ressources sont disponibles. En d'autres termes, il semble inutile de refaire des erreurs sur le problème d'ordonnancement sous-jacent alors qu'une solution est déjà presque entièrement déterminée : nous avons donc tenté d'améliorer l'heuristique GS en utilisant les

dates déterminées à partir de σ_∞ comme dates minimales de début pour toutes les opérations (et plus une partie seulement). Avec cette modification, le recouvrement est nettement amélioré, et GS atteint alors la borne inférieure dans $99,6\%$ des cas ! ! Cela semble plus naturel si nous considérons le fait que la quantité de ressources est quasi « illimitée » par rapport à la taille du graphe (nous avons pu vérifier que dans 96.2% de ces tests, les ressources sont en nombre suffisant pour réaliser σ_∞ sans aucun changement).

Revenons à ClockEdge, nous avons réalisé qu'en éliminant tous les graphes acycliques en entrée, la performance de l'heuristique remonte jusqu'à 93.6%. Ceci est surprenant puisque, dans un graphe acyclique, il est possible de donner par *retiming* un poids strictement positif à tous les arcs. Autrement dit, les graphes acycliques n'ont plus aucun arc de poids nul lors de la compaction ! Une analyse plus poussée des exemples nous a alors permis de comprendre le problème des cas acycliques : sur ces exemples, lorsque l'*intervalle de lancement* est inférieur au délai des opérations à ordonnancer, la solution optimale correspond souvent à des poids supérieurs à 1 sur les arcs, et donc à de grosses valeurs de *retiming*. Trouver un *retiming* qui transforme les arcs de poids nul en arcs de poids 1 n'est alors plus suffisant, et tous les cas où l'*intervalle de lancement* est très petit ne sont pas résolus de façon satisfaisante. Une solution est d'augmenter les effets du *retiming* entre composantes fortement connexes (en utilisant un algorithme similaire à celui présenté en section 2.4.1) et d'essayer de revenir en arrière[10] autant que possible une fois l'ordonnancement réalisé, afin de ne pas trop augmenter la taille du prologue/épilogue ainsi que la valeur de *maxlive*. En utilisant cette technique, la borne inférieure est alors atteinte dans $93,5\%$ des cas, graphes acycliques inclus.

Les résultats obtenus en incluant l'ensemble de ces optimisations sont présentés en table 2.7. Nous noterons toutefois que ces optimisations n'ont aucune incidence sur les résultats présentés dans le cas d'une simple unité fonctionnelle pipelinée, car dans ce dernier cas les ressources sont remplies facilement et le recouvrement n'a pas un rôle très important.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	24.4	24.2	23.7	23.6	23.5
Tests meilleurs que la Comp	100%	90.8%	94.5%	97%	100%
Tests strictement meilleurs que la Comp	0%	14.1%	18.5%	19.4%	21.6%
Gain moyen sur la Comp	0	0.184	0.678	0.779	0.916
Tests atteignant la borne inférieure	78.3%	82.9%	89.8%	93.5%	99.6%
Distance moyenne à la borne inférieure	0.92	0.736	0.242	0.141	0.004
Distance maximale à la borne inférieure	25	15	10	6	1
Valeur moyenne de *maxlive*	6.01	6.04	8.5	8.58	6.13
Valeur moyenne de Δr	0.806	1.01	2.51	2.55	0.95
Temps d'exécution (*retiming*)	0ms	1.3ms	0.21ms	0.79ms	0ms
Temps d'exécution (scheduling)	6.59ms	3.58ms	4.86ms	9.28ms	3.09ms

TAB. 2.7 – GS amélioré, et un meilleur *retiming* pour les cas acycliques de ClockEdge.

Nous avons pu remarquer qu'il était nécessaire de « pousser » beaucoup plus les opérations dans le cas de la compaction afin d'obtenir un recouvrement correct, contrairement au cas de ClockEdge et GS où bien souvent le recouvrement ne joue que sur une fraction du délai d'une opération. Ceci s'explique par la nature de l'approche décomposée : la complexité du pipeline (logiciel) à réaliser est traitée par la phase de décalage, et le problème résultant se limite réellement au corps de la boucle.

[10] Ce renversement se fait une fois de plus à l'aide de notre heuristique naïve, déjà évoquée précédemment, permettant de réduire *maxlive*.

Ici, le recouvrement ne sert qu'à compenser un petit déséquilibre lorsqu'une opération « dépasse ». Ainsi, à complexité égale, nous pouvons passer plus de temps à trouver un bon ordonnancement acyclique. Bien sûr, l'heuristique poussant les opérations est bénéfique à toutes les approches à cause du déséquilibre induit par l'ordonnancement au plus tôt, mais une technique simple de poussée est suffisante pour ClockEdge et GS qui produisent déjà un corps de boucle compact. Néanmoins, améliorer ces techniques de recouvrement reste un bon sujet de travaux futurs.

Il est clair que l'utilisation de ces techniques de pipeline logiciel en deux temps à base de *retiming* n'est pas justifiée lorsque les ressources ne sont pas limitées, puisque le problème peut alors être résolu de manière optimale en temps polynomial. Dans des cas de ressources limitées mais tout de même abondantes, l'heuristique GS a donc de bonnes chances de donner d'excellents résultats (optimaux la plupart de temps). Intéressons-nous maintenant à des cas où les ressources sont nettement plus limitées et commencent à poser un problème. Supposons que nos opérations de délai 20 bloquent une ressource unique du système (un *load* par exemple peut bloquer le bus mémoire lors de sa résolution, comme c'est le cas avec la vieille technologie de mémoire asynchrone dont la dernière incarnation a été la mémoire EDO). Dans ce cas, deux opérations de délai 20 ne peuvent plus être recouvertes. Corsons un peu la difficulté en augmentant le nombre de sommets à 20 (soit deux opérations de délai 20 dans le graphe) et observons les résultats présentés en table 2.8. Comme nous pouvons le constater, la domination de GS n'est plus écrasante, et ClockEdge reprend l'avantage, suivie de toutes les autres heuristiques à base de *retiming*.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	45	43.3	43.2	42.9	43.6
Tests meilleurs que la Comp	100%	97.4%	93.1%	97.5%	97.9%
Tests strictement meilleurs que la Comp	0%	31%	33.3%	35.6%	25.3%
Gain moyen sur la Comp	0	1.66	1.77	2.05	1.41
Tests atteignant la borne inférieure	53.5%	77.9%	77.5%	82.3%	72.9%
Distance moyenne à la borne inférieure	3.14	1.48	1.37	1.09	1.73
Distance maximale à la borne inférieure	23	20	21	20	20
Valeur moyenne de *maxlive*	11.2	12	17.5	17.5	11.8
Valeur moyenne de Δr	0.503	1.16	3.17	3.15	0.993
Temps d'exécution (*retiming*)	0ms	6.35ms	0.72ms	3.31ms	0ms
Temps d'exécution (scheduling)	27.8ms	19.8ms	27.8ms	49.1ms	21.4ms

TAB. 2.8 – Avec des opérations de délai 20 bloquant une ressource commune.

Note : dans une telle configuration, il est fort possible que l'*intervalle de lancement* optimal soit très éloigné de la borne inférieure (à cause des interférences entre les contraintes de ressources et les contraintes de dépendance).

Considérons un autre exemple avec la machine VLIW de départ, mais en supposant que toutes les opérations accèdent à une ressource commune durant leur dernier cycle (un bus d'accès aux registres par exemple), et toujours 20 sommets. Là encore les résultat sont édifiants (voir table 2.9) : la performance de GS s'effondre complètement. Seule ClockEdge atteint la borne inférieure dans plus de la moitié des cas avec 61.9%, contre 19.5% pour GS !

En continuant à changer les paramètres dans ce sens, des remarques similaires peuvent être faites : dès que les ressources posent une difficulté, les performances de l'heuristique GS améliorée disparaissent. Afin de savoir si la cause de ce phénomène était notre amélioration, nous avons

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	38.9	32.5	34.2	32.1	33.9
Tests meilleurs que la Comp	100%	99.6%	85.4%	98%	96.3%
Tests strictement meilleurs que la Comp	0%	84.6%	76.5%	88%	73.5%
Gain moyen sur la Comp	0	6.32	4.61	6.73	4.91
Tests atteignant la borne inférieure	8.8%	41.8%	36%	61.9%	19.5%
Distance moyenne à la borne inférieure	7.26	0.945	2.66	0.534	2.35
Distance maximale à la borne inférieure	39	6	31	5	16
Valeur moyenne de *maxlive*	11.2	12.2	16.2	16	12
Valeur moyenne de Δr	0.428	1.24	2.77	2.77	0.966
Temps d'exécution (*retiming*)	0ms	6.45ms	0.67ms	3.37ms	0ms
Temps d'exécution (scheduling)	23.3ms	12.1ms	19.5ms	35.6ms	16.4ms

TAB. 2.9 – Avec des opérations utilisant une ressource commune sur leur dernier cycle.

repris les mêmes tests avec l'heuristique GS non améliorée, et même avec l'heuristique GS sans aucune date minimale de départ. Dans les deux cas les performances sont encore moins bonnes. Ceci semble donc suggérer que le choix de *retiming* fait par cette heuristique est particulièrement mauvais lorsque la difficulté provient des contraintes de ressources. Notons cependant que, vu la performance de GS lorsque les ressources sont « quasi illimitées », la meilleure solution reste sans doute d'utiliser l'heuristique la plus appropriée au problème à résoudre, voire d'utiliser les deux heuristiques et de garder le meilleur résultat !

2.5.3 Cas irréalistes

Considérons maintenant un ensemble de cas limites peu réalistes, mais illustrant le comportement des heuristiques et appuyant nos remarques précédentes. Tout d'abord, afin de dissiper tout doute quant au modèle de machine, nous avons repris la machine VLIW de la table 2.4, mais nous l'avons modifiée afin que ses unités fonctionnelles ne soient plus pipelinées. Comme nous pouvons le

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	25.3	24.6	24.5	24.4	24.6
Tests meilleurs que la Comp	100%	92.3%	90.3%	93.9%	92.5%
Tests strictement meilleurs que la Comp	0%	21.6%	25.4%	26.9%	22.1%
Gain moyen sur la Comp	0	0.726	0.83	0.956	0.737
Tests atteignant la borne inférieure	66.3%	79.4%	77.6%	83.3%	80.7%
Distance moyenne à la borne inférieure	1.36	0.634	0.53	0.404	0.623
Distance maximale à la borne inférieure	22	8	10	7	11
Valeur moyenne de *maxlive*	5.89	6.07	7.93	8	6.07
Valeur moyenne de Δr	0.64	0.998	2.19	2.25	0.841
Temps d'exécution (*retiming*)	0ms	1.37ms	0.39ms	0.79ms	0ms
Temps d'exécution (scheduling)	7.97ms	4.63ms	8.93ms	14.4ms	5.58ms

TAB. 2.10 – VLIW non pipelinée.

constater sur les résultats présentés en table 2.10 pour 10 sommets, ClockEdge est alors la meilleure heuristique. Notons que pour cet exemple toutes les heuristiques ont de bonnes performances, mais

le cas est plus simple car les ressources sont moins complexes à gérer et, vue la valeur de l'*intervalle de lancement*, les opérations ont moins tendance à chevaucher plusieurs itérations. Ceci montre bien que le choix d'une machine pipelinée dans les exemples précédents était motivé par souci de réalisme et non pour faire paraître ClockEdge comme meilleure qu'elle ne l'est réellement. Comme nous pouvions nous en douter, le même test avec 20 sommets montre une suprématie encore plus grande de ClockEdge ; nous ne présentons pas les résultats afin de ne pas surcharger ce document.

Revenons donc maintenant au cas difficile qui nous intéresse, c'est-à-dire notre machine VLIW pipelinée. Jusqu'alors, nous n'avions considéré que des codes réalistes en termes de dépendances : beaucoup de dépendances indépendantes de la boucle synonymes de calcul dans l'itération, et quelques dépendances portées par la boucle réalisant une petite récurrence. Si maintenant nous considérons une récurrence plus profonde, avec 80% d'arcs de poids 1 (au lieu de 33% précédemment), 20% d'arcs de poids nul (au lieu de 67% précédemment) et 20 sommets, nous nous plaçons dans un cas beaucoup moins réaliste mais révélateur d'un défaut d'une partie des heuristiques de pipeline logiciel décomposé. Les résultats obtenus sont présentés en table 2.11. Le principal problème ici est

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	18.5	16.5	16.6	15.8	14.1
Tests meilleurs que la Comp	100%	95.9%	92.1%	96.6%	100%
Tests strictement meilleurs que la Comp	0%	44.6%	42.4%	55%	69.8%
Gain moyen sur la Comp	0	2	1.84	2.7	4.37
Tests atteignant la borne inférieure	29.9%	42.3%	40.8%	48.9%	96.4%
Distance moyenne à la borne inférieure	4.41	2.41	2.57	1.71	0.037
Distance maximale à la borne inférieure	18	15	14	13	2
Valeur moyenne de *maxlive*	15.5	15.8	18.4	18.8	16
Valeur moyenne de Δr	0.504	0.923	1.7	1.85	1.17
Temps d'exécution (*retiming*)	0ms	6.56ms	0.38ms	2.96ms	0ms
Temps d'exécution (scheduling)	14.7ms	9.83ms	13.6ms	24.3ms	6.74ms

TAB. 2.11 – Code avec récurrence profonde.

la vision limitée de l'approche décomposée. En effet, dans cette approche, toutes les dépendances portées par la boucle sont simplement coupées et considérées comme résolues par la séparation de l'ordonnancement en itérations successives. Pourtant, dans une optique de parallélisation efficace le recouvrement des motifs successifs doit rentrer en jeu, or couper toutes les dépendances portées par la boucle fait perdre de l'information utile au recouvrement. Le gros problème ici est la présence d'opérations de délai élevé sur les cycles de dépendance. En effet, l'*intervalle de lancement* moyen est de $14,1$ pour GS, c'est-à-dire bien inférieur au délai de la plus longue opération qui est de 20. Il en découle que certaines opérations vont se prolonger sur plusieurs itérations. Dans cette situation, les arcs portés par la boucle commencent à prendre toute leur importance puisque, pour recouvrir au mieux, il faut qu'une partie des opérations empiète largement sur l'itération suivante. Cependant nous pouvons constater que, même si l'optimal est peu atteint par ClockEdge, la distance à l'optimal reste très raisonnable, et le résultat reste donc d'une qualité appréciable. Si nous remplaçons les opérations de délai 20 par des opérations de délai 2, le phénomène doit logiquement disparaître puisque le recouvrement est alors nettement moins contraint par les dépendances portées par la boucle. C'est ce que nous pouvons constater en table 2.12, ClockEdge revient à un niveau comparable à GS.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	7	6.22	6.64	6.15	6.12
Tests meilleurs que la Comp	100%	99%	91.9%	99.7%	99.7%
Tests strictement meilleurs que la Comp	0%	54.5%	33.9%	59.2%	60.1%
Gain moyen sur la Comp	0	0.773	0.352	0.85	0.873
Tests atteignant la borne inférieure	35.8%	82.3%	54.6%	89.8%	92.1%
Distance moyenne à la borne inférieure	0.953	0.18	0.601	0.103	0.08
Distance maximale à la borne inférieure	6	2	5	2	2
Valeur moyenne de *maxlive*	15.8	16.8	18.2	18.8	15.8
Valeur moyenne de Δr	0.38	1.06	1.4	1.6	0.767
Temps d'exécution (*retiming*)	0ms	4.38ms	0.58ms	3.76ms	0ms
Temps d'exécution (scheduling)	5.47ms	4.32ms	4.84ms	9.58ms	5.71ms

TAB. 2.12 – Sans longues opérations.

L'heuristique GS au contraire fonctionne parfaitement dans le cas présenté en table 2.11 simplement parce qu'elle bénéficie des dates de l'ordonnancement optimal en ressources infinies, c'est-à-dire du meilleur placement possible pour former un corps aussi compact que possible (donc, un recouvrement quasi-optimal). En outre, vu la grande disponibilité des ressources dans cette situation, adapter l'optimal n'est pas très difficile. Pour confirmer notre interprétation, nous pouvons enlever notre amélioration à GS : nous ne donnons plus de date de départ minimale à toutes les opérations, mais seulement à certaines (suivant l'idée de Gasperoni et Schwiegelshohn rappelée précédemment). De façon logique, nous perdons alors toute information pour le recouvrement optimal et nous obtenons la table 2.13, où GS offre des résultats pires que ClockEdge.

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	18.5	16.5	16.6	15.8	16
Tests meilleurs que la Comp	100%	95.9%	92.1%	96.6%	90.7%
Tests strictement meilleurs que la Comp	0%	44.6%	42.4%	55%	52.3%
Gain moyen sur la Comp	0	2	1.84	2.7	2.47
Tests atteignant la borne inférieure	29.9%	42.3%	40.8%	48.9%	42.3%
Distance moyenne à la borne inférieure	4.41	2.41	2.57	1.71	1.94
Distance maximale à la borne inférieure	18	15	14	13	10
Valeur moyenne de *maxlive*	15.5	15.8	18.4	18.8	16.4
Valeur moyenne de Δr	0.504	0.923	1.7	1.85	1.19
Temps d'exécution (*retiming*)	0ms	6.22ms	0.57ms	3.39ms	0ms
Temps d'exécution (scheduling)	14.9ms	10.2ms	13.4ms	24.7ms	9.23ms

TAB. 2.13 – GS non amélioré.

Ceci confirme donc bien la suprématie de GS amélioré lorsque les ressources ne créent pas de difficulté. Cependant, cette force est aussi une faiblesse puisque GS n'offre aucune liberté ni au niveau du *retiming* ni au niveau de l'ordonnancement. Ainsi, l'heuristique est totalement enfermée dans son modèle et n'est pas à même de traiter les cas où les ressources sont limitées, ou, plus généralement, les cas où l'ordonnancement optimal sans contraintes de ressources est très loin de la meilleure solution. Si nous reprenons notre modèle de machine pour lequel les opérations accèdent à une ressource commune lors de leur dernier cycle (déjà présenté en table 2.9), toujours avec le

code non réaliste rempli de dépendances portées par la boucle (et GS amélioré), nous obtenons les résultats présentés en table 2.14. Là encore, ce comportement de « section critique » est une nouvelle

	Comp	Clock	MinEdge	ClockEdge	GS
Intervalle de lancement moyen	22.6	21	21.8	20.9	21.9
Tests meilleurs que la Comp	100%	97.6%	89.3%	96.6%	80.1%
Tests strictement meilleurs que la Comp	0%	53.7%	42.3%	60.1%	32.1%
Gain moyen sur la Comp	0	1.57	0.767	1.68	0.674
Tests atteignant la borne inférieure	32.9%	66.9%	54.1%	76.6%	37.2%
Distance moyenne à la borne inférieure	1.93	0.356	1.16	0.247	1.26
Distance maximale à la borne inférieure	28	6	26	3	8
Valeur moyenne de *maxlive*	17.4	17.8	18.5	18.6	17.6
Valeur moyenne de Δr	0.281	0.855	1.19	1.35	0.768
Temps d'exécution (*retiming*)	0ms	6.29ms	0.46ms	3.08ms	0ms
Temps d'exécution (scheduling)	18.9ms	14.6ms	19ms	35.8ms	19.8ms

TAB. 2.14 – Les opérations accèdent à une ressource commune lors de leur dernier cycle.

contrainte inattendue (et plutôt dure) et GS n'est plus à même de donner d'aussi bons résultats, ce qui confirme également le peu de robustesse de l'heuristique.

2.6 Conclusions

L'heuristique de Calland, Darte et Robert [10] est fondée sur l'utilisation de *retiming* sur le graphe de dépendance de la boucle, afin de rendre une compaction de cette dernière plus efficace. L'heuristique originale utilise l'algorithme de Leiserson et Saxe [70] afin de produire par *retiming* un graphe dont le chemin critique pour la compaction est minimal. Cette optimisation permet alors de donner des garanties de performance du résultat final relativement à l'optimal dans le cas de ressources homogènes (le cas le plus simple pour les ressources). Nous avons proposé dans ce chapitre d'étendre cet algorithme à un modèle plus général décrivant de manière distincte les latences entres opérations, les temps de réservation des ressources, et les temps de calcul. En particulier, ce modèle étendu permet une représentation réaliste d'une boucle exécutée sur un processeur pipeliné. Calland, Darte et Robert proposèrent également de combiner cette minimisation à une minimisation du nombre de contraintes restant pour la compaction, et donnèrent une solution à ce nouveau problème sous forme de programme linéaire. Nous avons proposé dans ce chapitre un algorithme de graphe résolvant le problème en temps polynomial que nous avons pu implanter lors de nos expérimentations.

Afin de réaliser un large nombre d'expérimentations, nous avons réalisé PASTAGA, une plate-forme de test permettant l'utilisation de diverses heuristiques de pipeline logiciel décomposé sur des échantillons de graphes de dépendance générés aléatoirement. Nous avons intégré au générateur un large nombre de paramètres permettant d'ajouter suffisamment de contraintes sur les graphes générés pour garantir leur réalisme. Grâce à cette cette plate-forme, nous avons pu comparer la compaction de boucle, l'heuristique de Calland, Darte et Robert avec ou sans notre amélioration, et l'heuristique de Gasperoni et Schwiegelshohn [48], nous avons également pu nous faire une idée de leur efficacité absolue. Nos expérimentations permettent de dégager deux principaux cas de figures :

– lorsque les ressources sont saturées (manque général de ressources pour exécuter en même temps toutes les opérations disponibles), notre heuristique se dégage comme nettement plus

efficace. Ceci confirme l'intuition que minimiser le nombre de contraintes pour la compaction lui facilite le travail ;
- lorsque les ressources ne sont pas saturées, le travail de la compaction est de toutes façons plus simple et seules les dépendances peuvent poser problème. Dans ce cas, l'heuristique de Gasperoni et Schwiegelshohn est nettement meilleure. Ceci se comprend aisément puisque cette heuristique est fondée sur une modification de l'ordonnancement en ressources infinies. Néanmoins, pour atteindre les meilleurs résultats, nous avons dû la modifier afin de la rendre encore plus similaire à cet ordonnancement optimal.

Le large nombre d'expérimentations effectué nous a également permis de mieux comprendre le fonctionnement de ces diverses heuristiques ainsi que leur faiblesses. Comme toutes les heuristiques, celles fondées sur le décalage/ordonnancement ne sont pas exemptes de défauts et peuvent tomber dans certains comportements pathologiques. Comme nous l'avons vu, la séparation en deux phases permet de simplifier la situation en transformant un problème cyclique en problème acyclique. Cependant, même si tout ordonnancement cyclique peut se décomposer en un *retiming* suivi d'une compaction, retrouver une solution optimale demande de trouver à la fois le bon *retiming* et la bonne compaction. Choisir un *retiming* minimisant le chemin critique ou le nombre de contraintes permet seulement de se rapprocher des bornes inférieures sur la performance, mais pour certains choix de tels *retimings* il n'existe tout simplement pas de compaction permettant d'atteindre l'optimal[11]. À l'inverse, une fois le *retiming* choisi, même si ce *retiming* correspond au *retiming* d'une solution optimale, la compaction doit encore résoudre un problème toujours NP-complet. De plus, comme nous l'avons vu lors des expérimentations, découper le problème ne fait pas perdre à la solution sa nature cyclique, aussi il est préférable d'inclure dans la compaction des heuristiques permettant un meilleur recouvrement des motifs. Ceci est d'autant plus difficile que la compaction n'a pas de compréhension des cycles de dépendance du graphe.

Néanmoins, la séparation en deux phases du problème de pipeline logiciel présente un certain nombre d'avantages. En particulier, elle permet de convertir la borne au pire cas pour l'ordonnancement acyclique en borne au pire cas pour le pipeline logiciel. Cette idée importante est due à Gasperoni et Schwiegelshohn et elle est un des intérêts majeurs du pipeline logiciel décomposé. Aucune autre méthode (pas même le *modulo scheduling*) ne propose une telle garantie : il n'a pas pu être déterminé dans quelle mesure elles s'approchent de l'optimal. En outre, cette séparation offre d'autres avantages d'un intérêt plus pratique : tout d'abord, une liberté totale est laissée quant au choix de l'algorithme de compaction, la prise en compte des ressources n'intervenant pas dans la phase de *retiming*. Lors de nos expérimentations, nous n'avons utilisé que des algorithmes simples et peu coûteux, à base d'ordonnancement par liste, modifiés pour tenir compte du recouvrement des motifs. Rien n'empêche cependant de mettre au point des algorithmes plus élaborés destinés à des architectures spécifiques ou effectuant une recherche plus complète. Un autre avantage est que le *retiming* permet de contrôler assez facilement le déplacement des opérations entre itérations simplement en ajoutant des arcs au graphe de dépendance. Ainsi il est tout à fait envisageable de limiter la quantité de décalage de la boucle dans le but de réduire la consommation en registres, ou encore de relâcher les contraintes de période d'horloge afin de laisser plus de liberté à l'algorithme de minimisation du nombre d'arcs de poids nul. Comme illustré par l'algorithme 2, n'importe quelle contrainte de poids minimal sur chaque arc ou chemin peut être ajoutée sous forme d'un arc supplémentaire

[11] Bien évidemment, il faut ici tempérer cette affirmation : une compaction autorisant des dates séparées par plus d'un *intervalle de lancement* permettrait de modifier profondément l'ordonnancement et de retrouver l'optimal, mais une telle solution correspond à un *retiming* différent. En d'autres termes, si nous nous limitons au modèle « *retiming* puis compaction sans recouvrement » alors nous pouvons effectivement mal choisir le *retiming* et ne plus pouvoir retrouver l'optimal.

dans le graphe.

Jusqu'alors, dans le cadre de nos expérimentations, nous n'avons comparé notre technique qu'avec d'autres techniques de pipeline logiciel décomposé. Toutefois, la comparaison avec les bornes inférieures classiques montre une bonne efficacité absolue. Évidemment, il manque une comparaison avec d'autres techniques comme le *modulo scheduling* et l'*Enhanced software pipelining*. Pour l'*Ehenced software pipelining* [80], nous n'avons pas jugé la comparaison très significative. En effet, le choix exact du décalage à appliquer lors de l'algorithme n'est pas précisément défini. Autrement dit, le décalage mis en œuvre a un aspect indéterministe, arbitraire, et la façon dont va procéder l'algorithme implanté en pratique va influer fortement sur le résultat. Dans ce contexte, il devient difficile d'interpréter les résultats puisque nous ne connaissons pas la fonction objective optimisée. Par ailleurs, il est impossible de savoir si l'algorithme s'arrête lorsqu'il atteint un minimum local ou si un objectif caché est bel et bien atteint. En outre, lorsque la compaction est contrainte par les dépendances, décaler les opérations situées aux extrémités du motif revient vraisemblablement à minimiser la période d'horloge, rien de plus.

Dans le cas du *modulo scheduling* le problème est différent. L'algorithme de *modulo scheduling* itératif décrit par Rau dans [93], sans doute le plus connu et le plus utilisé, est davantage une méthodologie qu'une procédure précise. Le principe est de commencer par fixer l'*intervalle de lancement* et, grâce à ce paramètre, d'ordonnancer les opérations suivant une certaine priorité[12] en calculant à chaque étape les dates au plus tôt et au plus tard entre lesquelles les opérations encore disponibles peuvent être placées dans le motif. Si cet ordonnancement échoue, il suffit alors d'augmenter l'*intervalle de lancement* choisi et de recommencer. L'indéterminisme commence réellement à apparaître lorsque B. Rau décrit la possibilité d'enlever une opération de l'ordonnancement pour laisser la place à une autre qui, sinon, n'aurait pu être ordonnancée. Ceci rend, en théorie, le *modulo scheduling* parfait en termes de performance si l'on s'autorise une quantité illimitée de tels réordonnancements. En d'autres termes, l'*iterative modulo scheduling* est optimal au prix d'une complexité exponentielle. En pratique, on utilise donc une quantité *BudgetRatio* de réordonnancements limitée. En fait, B. Rau décrit plus une manière de passer d'un problème inabordable à un problème soluble. Mais les performances et la qualité de la méthode restent fortement dépendantes du choix de la fonction de priorité et du *BudgetRatio* pour une architecture et un problème donnés. Comme B. Rau le décrit dans [93] : « Un *BudgetRatio* trop petit va entraîner l'essai de valeurs d'*intervalle de lancement* croissantes jusqu'à ce qu'un ordonnancement soit trouvé à une valeur plus grande que nécessaire.[...]D'autre part, une fois que le *BudgetRatio* a été augmenté suffisamment pour atteindre l'*intervalle de lancement* minimum, toute augmentation subséquente ne peut améliorer la qualité de l'ordonnancement.[...]Augmenter le *BudgetRatio* signifie seulement que davantage de temps de compilation sera consacré à des tentatives destinées à échouer. Ceci suggère la possibilité qu'il existe une valeur optimale du *BudgetRatio* pour laquelle le résultat est proche de l'optimal et la complexité est également proche de sa valeur minimale. »[13]. Une comparaison significative avec le *modulo scheduling* nécessiterait donc une étude des paramètres optimaux pour chaque situation, ou mieux encore une étude des paramètres les plus adaptés à l'ensemble des situations possibles. Nous avons considéré qu'une telle étude, bien qu'intéressante, n'était pas adaptée au cadre mathématiquement bien

[12] notons que le choix de la fonction de priorité va fortement influer sur le succès de l'ordonnancement dans l'*intervalle de lancement* choisi.

[13] *Too small a BudgetRatio results in having to try successively larger values of II until a schedule is found at a larger II than necessary.[...]On the other hand once the BudgetRatio has been increased enough that the minimum achievable II has been reached, further increasing BudgetRatio cannot be beneficial in terms of schedule quality.[...]Increasing BudgetRatio only means that more compile time is spent on attempts that are destined to be unsuccessfull. This suggests the possibility that there is some optimum value of BudgetRatio for which the execution time is near optimal and the comutational complexity is also near its minimal value.*

défini du pipeline logiciel décomposé, notre but étant d'abord de comprendre les mécanismes d'interaction entre *retiming* et compaction, et d'identifier les objectifs les plus pertinents lors du calcul du *retiming*. C'est pourquoi nous avons choisi de laisser la comparaison avec le *modulo scheduling* pour de futurs travaux.

Remarquons tout de même qu'aussi bien le comportement que la complexité du *modulo scheduling* restent difficiles à évaluer, et il semble légitime de se demander dans quelle mesure il emploie la bonne stratégie de recherche d'une solution. Bien sûr, une fois les bons paramètres déterminés, il est sûrement tout à fait efficace en pratique. Cependant, nos expérimentations semblent suggérer qu'il n'est pas nécessairement utile de chercher une solution générale et universelle à l'ensemble des situations. Comme nous l'avons vu, lors de situations dans lesquelles les ressources sont suffisantes, un algorithme proche de l'ordonnancement en ressources infinies comme celui de Gasperoni et Schwiegelshohn atteint presque immanquablement un résultat optimal. À l'inverse lorsque les ressources sont trop peu nombreuses ou posent des contraintes trop complexes, un algorithme spécifiquement conçu pour faciliter la compaction donne d'excellents résultats. Dans les deux cas, la méthode employée est totalement déterministe et polynomiale, en deux étapes et sans retour en arrière. Ici, pas de compromis à faire entre temps de calcul et efficacité. Chacune des heuristiques s'attaque au cas de figure pour lequel elle est spécialisée. Pour finir, nous citerons quelques exemples appuyant notre propos :

- dans l'Itanium d'Intel [32, 59], la quantité de ressources est cachée dans le processeur (pour que l'évolution dans les futures versions se fasse de manière transparente), et le langage machine accessible exprime le parallélisme sous forme de séquences d'opérations parallèles. Dans une telle situation, deux approches sont possibles : faire du pipeline logiciel en ressources infinies afin de produire un code efficace sur toutes les versions de ce processeur, ou faire du pipeline logiciel pour une version spécifique en optimisant le code pour un nombre précis de ressources. Les deux situations correspondent respectivement à l'heuristique de Gasperoni et Schwiegelshohn et à notre heuristique ;
- le processeur LANai 3.0 développé par Myricom [81] est un processeur utilisé dans les contrôleurs de cartes Myrinet, possédant une seule unité fonctionnelle et un pipeline court. Pour ce processeur, les cycles sont explicitement exprimés dans le programme par l'ajout de NOP. C'est un cas de figure dans lequel les ressources sont très limitées et pour lequel notre méthode peut s'avérer payante (notre exemple utilisait un modèle d'architecture similaire à ce processeur) ;
- le compilateur PICO développé aux HP Labs [98] synthétise une architecture à partir d'une description comportementale en C. Dans sa dernière étape, il nécessite l'ordonnancement des opérations à des cycles précis afin de générer un motif à la fois efficace et nécessitant la synthèse d'un minimum de ressources. Dans ce cas, l'utilisation de notre approche permettrait de se concentrer sur un problème acyclique dans lequel les contraintes ont été minimisées afin de faciliter la répartition des opérations dans le motif (et entraîner la synthèse d'un minimum de ressources).

Chapitre 3

Réordonnancement par étages

3.1 Introduction

Comme nous l'avons vu au chapitre 2, les processeurs modernes offrent de plus en plus de capacités de parallélisme (aussi bien sous la forme d'unités fonctionnelles multiples que pipelinées), capacités dont cherchent à tirer parti les algorithmes de pipeline logiciel. Malheureusement, ces algorithmes, bien que permettant souvent de regrouper une grande quantité d'opérations sur un faible nombre de cycles du processeur, ont également le défaut fréquent d'augmenter considérablement la quantité de registres nécessaire pour stocker les valeurs intermédiaires calculées lors de l'exécution de la boucle. Le problème est que, lorsque le processeur cible ne dispose plus d'assez de registres pour réaliser tel quel l'ordonnancement déterminé par l'algorithme de pipeline logiciel, il est nécessaire de recourir à des échanges mémoire (aussi appelés *spill code*) qui dégradent considérablement la performance du résultat final. Intuitivement, cette augmentation des besoins en registres est de toutes façons logique, et dans une certaines mesure inévitable. En effet, plus les opérations sont exécutées en parallèle, plus les résultats intermédiaires produits simultanément sont nombreux. De plus, le pipeline logiciel lui-même aggrave ce phénomène en étalant les chaînes d'opérations dépendantes sur plusieurs itérations. Ainsi, l'objectif n'est pas tant d'essayer d'éviter une surconsommation en registres que d'essayer de limiter cette surconsommation. En conséquence, la plupart des travaux considérant la consommation mémoire dans le cadre du pipeline logiciel sont consacrés à la minimisation du nombre de registres requis pour l'exécution d'une boucle, soit en choisissant un meilleur ordonnancement, soit en tentant d'optimiser un ordonnancement donné.

Mais avant de minimiser ce nombre de registres, encore faut-il le connaître. Une borne inférieure à ce nombre est donnée par le nombre maximal de valeurs « en vie » à tout instant de l'ordonnancement par R. A. Huff [57], elle est désignée sous le nom de *maxlive*. Atteindre cette borne est possible, mais peut nécessiter soit un support matériel sous la forme d'une queue de registres tournante (*rotating register file*, comme dans le processeur Cydra 5 [29] ou l'Itanium d'Intel [32, 59]), soit un déroulage de la boucle ou un renommage des registres (voir [37, 71] pour un aperçu de toutes ces allocations optimales). Le problème est que les techniques de déroulage ou de renommage réduisent l'efficacité du code, respectivement en augmentant sa taille et en ajoutant des instructions dans le corps de la boucle. Parfois, il est préférable d'utiliser une allocation suboptimale plutôt que de subir ces effets. Cependant, dans [95], de nombreuses expérimentations montrent que même une allocation non optimale atteint la borne dans une grande majorité des cas et ne la dépasse que d'un registre dans presque tous les autres. Autrement dit, minimiser *maxlive* dans le but de réduire le nombre de registres requis par une boucle est tout à fait pertinent.

Les premiers travaux dans ce domaine ont cherché à intégrer *maxlive* comme critère supplé-

mentaire lors de l'ordonnancement. La première étape pour qu'un algorithme de pipeline logiciel minimise *maxlive* tout en conservant des objectifs de performance est d'être capable d'exprimer précisément le coût d'un ordonnancement à la fois à l'aide de son *intervalle de lancement* et de *maxlive*. Les valeurs « en vie » à un instant donné d'un ordonnancement définissent *maxlive*. Elles correspondent à une partie des dépendances de flot présentes dans l'ordonnancement (en effet, une valeur créée par une opération pour être utilisée par une opération future correspond à une dépendance de type : écriture suivie de lecture). Q. Ning et G. R. Gao proposent dans [82] une analyse détaillée du nombre de valeurs en vie associé à une variable. Ils expriment ce nombre sous forme d'une quantité non linéaire dépendant de l'ordonnancement et des dépendances entre instructions. Le problème est que cette expression étant non linéaire, Q. Ning et G. R. Gao ne parviennent pas à l'utiliser telle quelle. Ils sont obligés d'en faire une approximation, qu'ils intègrent au coût d'un programme linéaire en nombres entiers déterminant l'ordonnancement optimal en ressources infinies. Ils montrent alors que le problème reste polynomial (rappelons que l'ordonnancement en ressources infinies est à la base un problème polynomial). Il en résulte un algorithme polynomial d'ordonnancement optimal sans contraintes de ressources minimisant une approximation de *maxlive*.

Cette première solution est étendue par R. Govindarajan, E.R. Altman et G.R. Gao dans [53] au cas de ressources limitées. Ils proposent dans cet article un algorithme d'ordonnancement optimal sous contraintes de ressources minimisant toujours la même approximation de *maxlive*. L'algorithme est également fondé sur la programmation linéaire en nombres entiers, mais cette fois, avec les contraintes de ressources, il n'est plus polynomial. Une formulation exacte par programmation linéaire en nombres entiers est finalement donnée par A. E. Eichenberger, E. S. Davidson et S. G. Abraham dans [35]. Ils réussissent à exprimer *maxlive* dans le programme sous forme de contraintes linéaires en remarquant que, pour un ordonnancement cyclique de période λ, le nombre minimal de registres requis varie dans le temps de façon périodique de période λ : il suffit donc de décomposer la valeur de *maxlive* sur un intervalle de taille λ. Le programme résultant contient alors λ contraintes par arc du graphe de dépendance (en plus des contraintes de précédence et de ressources). Sans surprises, il devient vite énorme et sa résolution n'est envisageable en pratique que sur de très petits exemples[1].

Une alternative est d'appliquer la même approche que pour le pipeline logiciel en général : le problème étant NP-complet, nous utilisons des heuristiques pour le résoudre. Suivant cette idée, plusieurs algorithmes de pipeline logiciel prenant en compte *maxlive* ont été proposés (voir par exemple [57, 76] fondés sur le *modulo scheduling*). Naturellement, le problème des heuristiques de pipeline logiciel est que justement elles ne résolvent pas le problème de façon exacte. Autrement dit, l'heuristique peut très bien passer à côté d'une solution optimale et produire un résultat de qualité moindre aussi bien en termes de performances qu'en termes du nombre de registres requis. Un compromis est donc généralement à trouver entre performance de l'ordonnancement, coût en registres et temps de calcul du résultat. Evidemment comme bien souvent, plus le nombre de paramètres à prendre en compte est grand lors de la résolution d'un problème NP-complet, plus la qualité du résultat est incertaine. Ainsi, il devient vite difficile de dire si un compromis donné est plus avantageux en raison de sa performance ou de son coût en registres, et il n'est pas toujours aisé de savoir quel paramètre a le plus de poids dans la qualité générale du résultat.

Une approche radicalement différente consiste à partir d'un ordonnancement donné et à en conserver les avantages les plus importants tout en le modifiant légèrement afin d'améliorer sa qualité globale. Pour cela, reconsidérons le problème du pipeline logiciel : comme nous l'avons vu en section 2, un ordonnancement cyclique se découpe de façon naturelle en une combinaison

[1] Nous avons intégré ce programme à l'intérieur de l'outil PASTAGA présenté en section 2.5, et nous avons pu constater que sa résolution en nombres entiers fait « planter » le solveur pip à partir d'environ 5 sommets!

d'un *retiming* sans contraintes de ressources et d'un ordonnancement acyclique sous contraintes de ressources. L'idée du *retiming* initial est de mieux organiser les dépendances dans le but de construire un motif pour le corps de notre boucle. L'ordonnancement acyclique construit alors le motif proprement dit en prenant en charge les contraintes de ressources. Rien ne nous empêche alors de penser à une troisième étape : un *retiming* supplémentaire afin de réorganiser au mieux les dépendances sans changer le motif du corps de la boucle. De cette manière, le motif produit par la phase d'ordonnancement acyclique est conservé : les contraintes de ressources restent satisfaites et la performance de l'ordonnancement reste inchangée. Même si peu de gains sont possibles de cette manière, il n'y pas de compromis à faire : la solution ne peut qu'être améliorée (ou au pire inchangée).

Bien que formulé autrement par A.E. Eichenberger et E.S. Davidson dans [33, 36], le réordonnancement par niveaux (ou *stage scheduling*) est la mise en œuvre de cette idée. Le problème se pose alors de la façon suivante : étant donné un motif pour le corps de notre boucle (donc un ordonnancement cyclique), trouver un décalage des instructions minimisant *maxlive*. Dans [33], Eichenberger et Davidson proposent un certain nombre d'heuristiques fondées sur le principe du réordonnancement par niveaux afin de réduire *maxlive*, tandis que dans [36] ils proposent une résolution exacte du problème par la résolution d'un nombre exponentiel (en fonction du degré du graphe) de programmes linéaires comportant chacun un nombre exponentiel de contraintes (nous reviendrons sur cette formulation en section 3.4.4). Bien qu'il précisent qu'en pratique le nombre réel de contraintes de leurs programmes est faible, ils ne donnent aucun moyen permettant d'éviter l'explosion combinatoire du problème. Ainsi, deux questions restent toujours ouvertes :

- quelle est la complexité du problème du réordonnancement par niveaux ?
- est-il possible d'exprimer le problème du réordonnancement par niveaux sous la forme d'un algorithme plus simple que celui de [36] ?

La suite de ce chapitre est organisée de la manière suivante : tout d'abord, en section 3.2, nous présentons un exemple illustrant le mécanisme du réordonnancement par niveaux nous laissant augurer le type d'amélioration qu'il est possible d'espérer. En section 3.3, nous détaillons la formulation du coût mémoire d'une boucle (*maxlive*) et nous démontrons quelques propriétés qui nous permettrons de calculer ce coût plus efficacement. En section 3.4, nous répondons aux deux questions posées précédemment : tout d'abord, nous prouvons que le réordonnancement par niveaux est un problème NP-complet au sens fort. Puis nous en donnons une formulation exacte sous forme de programme linéaire en nombres entiers comportant un nombre polynomial de contraintes (en la taille du graphe), et nous expliquons comment nous avons pu améliorer la formulation proposée par Eichenberger, Davidson et Abraham dans [36]. Enfin, nous proposons un algorithme polynomial d'approximation du réordonnancement par niveaux (avec rapport d'approximation) avant de conclure en section 3.6.

3.2 Exemple

Étudions sur un exemple le mécanisme mis en œuvre par le réordonnancement par niveaux. Considérons l'exemple représenté en figure 3.1. Dans cet exemple, la situation est très simple : seulement trois opérations ordonnancées séquentiellement sans recouvrement (l'*intervalle de lancement* vaut la longueur du motif : 3), le motif correspondant au corps de la boucle est également représenté. Ici, nous supposons que toutes les dépendances portent des valeurs, c'est-à-dire que chaque arc du graphe correspond à une valeur temporaire, créée par le résultat d'une opération, qui devra être stockée dans un registre avant d'être consommée comme opérande de l'opération destination.

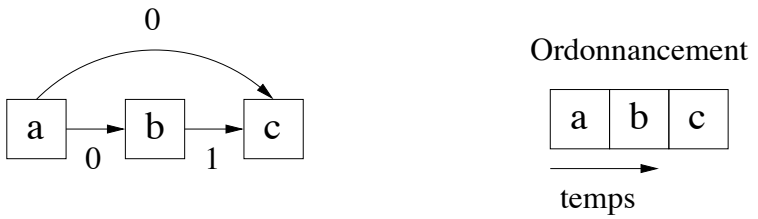


FIG. 3.1 – Exemple de graphe avec le motif répété à chaque itération.

Afin de mieux voir l'évolution dans le temps des valeurs calculées à chaque itération, nous avons également représenté une partie du graphe de dépendance étendu de la boucle (voir figure 3.2). Dans ce graphe étendu, chaque sommet correspond à une opération particulière d'une itération de la boucle, chaque arc a une valeur effectivement calculée et stockée jusqu'à sa consommation, et le temps progresse horizontalement de gauche à droite. Tous les arcs sortant d'un même noeud correspondent au même résultat et donc à une même valeur stockée dans un registre. Une valeur a besoin d'être stockée et est dite « en vie » tant que son dernier consommateur n'a pas été exécuté. Comme nous pouvons le constater sur la figure 3.1, sur cet exemple, lors de toutes les itérations

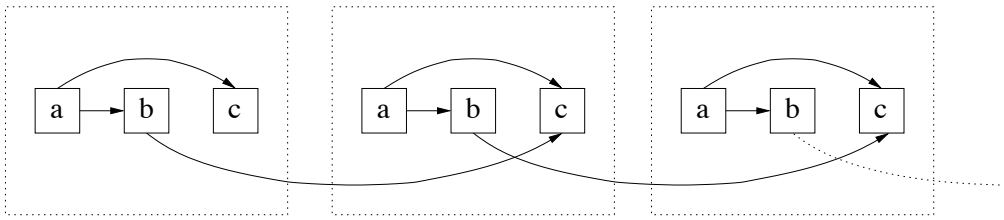


FIG. 3.2 – Version étendue de l'exemple de la figure 3.1.

après la première, il y a toujours trois valeurs en vie après l'exécution de b et avant celle de c dans la même itération : la valeur de b de l'itération précédente et celle de a de l'itération courante nécessaires pour le calcul de c, ainsi que la valeur de b de l'itération courante nécessaire pour la prochaine itération.

Pourtant, en appliquant un *retiming* bien choisi, il est possible de faire décroître ce nombre maximal de valeurs en vie simultanément. L'idée est de retarder l'exécution de toutes les opérations b d'une itération : ainsi, à l'itération i, nous exécuterons l'opération b qui était exécutée jusqu'alors à l'itération $i-1$, et nous obtenons le graphe étendu de la figure 3.3. Comme nous pouvons le

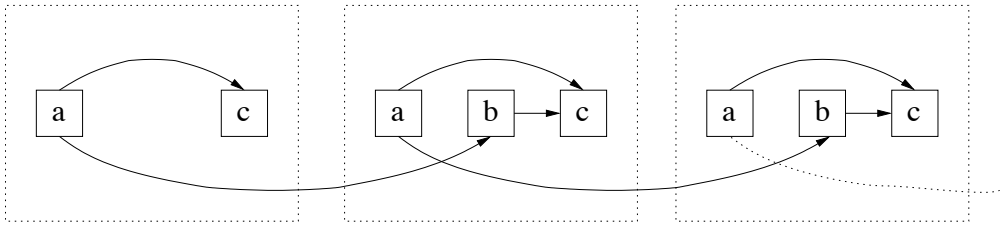


FIG. 3.3 – Solution après *retiming*.

constater, le code résultant n'a jamais besoin de stocker plus de deux valeurs simultanément pour le calcul de la boucle. En pratique, si une telle amélioration est possible en plusieurs endroits du graphe, il n'est pas difficile d'imaginer que le gain peut devenir gros. Les résultats que nous pouvons

trouver dans [35, 36] ne font que confirmer cette supposition : ils démontrent expérimentalement qu'une amélioration moyenne de 24% peut être réalisée sur un ensemble de *programmes étalons* classiques (amélioration obtenue en ajoutant le réordonnancement par étages à la suite du pipeline logiciel lors de la compilation).

3.3 Quantité de mémoire requise par une boucle

Nous utilisons, comme à l'habitude, un graphe de dépendance $G = (V, E)$ afin de représenter une boucle. V est l'ensemble des instructions du corps de la boucle et E l'ensemble des relations de dépendance entre instructions. Ce graphe est pondéré sur ses arcs par d, la distance de dépendance entre deux instructions et l, la latence entre deux opérations. Par la suite, nous ferons une distinction entre deux types d'arcs dans G : les arcs correspondant à une valeur intermédiaire devant être stockée dans un registre avant d'être consommée comme opérande d'une ou plusieurs opérations (une partie des dépendances de flot), et les arcs ne nécessitant pas de stockage intermédiaire (les dépendances de flot portant sur la mémoire globale et les autres dépendances : anti-dépendances et dépendances de sortie). Nous appelons E_{val} le sous-ensemble de E correspondant aux arcs nécessitant un stockage intermédiaire et $E_{dep} = E - E_{val}$. De même, tous les sommets ne produiront pas un résultat intermédiaire devant être stocké dans un registre, mais seulement ceux ayant un arc sortant dans E_{val}. Nous définissons donc $V_{val} = \{u \in V \mid \exists v \in V, e = (u, v) \in E_{val}\}$ cet ensemble de sommets.

De même que dans la section 2.2, nous définissons un ordonnancement σ comme une fonction de $V \times \mathbb{Z}$ dans $\mathbb{Z}$ qui assigne à chaque opération $(v, i) \in V \times \mathbb{Z}$ une date d'exécution $\sigma(v, i)$. Nous nous intéressons aux ordonnancements cycliques, c'est-à-dire pouvant être exprimés sous la forme $\sigma(v, i) = a_v + \lambda i$ et également appelés ordonnancements modulo (car deux opérations (v, i) et (v, i') correspondant à la même instruction dans deux itérations différentes sont exécutées à la même date modulo λ). Rappelons que λ est appelé *intervalle de lancement* de l'ordonnancement et correspond au temps séparant le début de deux itérations successives. Si notre graphe subit un *retiming* une fois un ordonnancement initial déterminé, alors ceci correspond à un nouvel ordonnancement $\sigma(v, i) = a_v + \lambda(i + r(v))$ pour l'opération (v, i) (par définition du *retiming*, v est retardé de $r(v)$ itérations).

Enfin, afin de gérer précisément l'espace de stockage intermédiaire, nous devons enrichir notre modèle de deux pondérations supplémentaires sur les arcs de E_{val}. En effet, selon l'architecture utilisée, les opérandes d'une opération ne sont pas décodés et récupérés au même étage dans le pipeline du processeur. Autrement dit, un certain nombre de cycles processeur s'écoulent entre le début d'une opération et la libération effective des registres. Nous notons ce nombre de cycles $t_{in}(e)$. Il est spécifique à chaque utilisation d'une valeur intermédiaire, donc à chaque arc, car tous les opérandes de l'opération destination ne sont pas nécessairement décodés au même cycle. De façon analogue, le résultat d'une opération exécutée par le processeur n'est généralement écrit de façon effective dans un registre que vers les derniers étages du pipeline. Le nombre de cycles séparant le début de l'exécution d'une opération de l'écriture du résultat dans un registre est une quantité spécifique à chaque valeur intermédiaire, donc à chaque arc (et non pas à chaque sommet si nous autorisons une opération à produire plusieurs résultats). Il dépend de l'architecture, du nombre d'étages du pipeline et du nombre de valeurs produites par l'opération, Nous notons ce nombre de cycles $t_{out}(e)$. Nos arcs E_{val} sont donc enrichis des deux pondérations t_{in} et t_{out}. De façon formelle à un instant t, pour tout arc $e = (u, v) \in E$, si $\sigma(u, i) + t_{out}(e) \leq t$ alors le résultat de l'opération (u, i) a été écrit dans un registre, et si $\sigma(v, i) + t_{in}(e) \leq t$ alors l'opération (v, i) a récupéré la valeur intermédiaire correspondant à e et n'en a donc plus besoin. Si cette dernière opération est la dernière dépendant de la valeur transportée par e, alors l'espace de stockage de cette valeur est libre.

De façon logique, nous imposons à une valeur d'être écrite dans un registre avant de pouvoir être récupérée, donc :

$$\forall e = (u, v) \in E_{val}, \sigma(u, i) + t_{out}(e) < \sigma(v, i) + t_{in}(e)$$

Cette propriété est toujours vérifiée par un ordonnancement correct si $t_{out}(e) - t_{in}(e) < l(e)$. Or, par définition, la latence d'un arc est le nombre minimum de cycles processeur devant séparer deux opérations dépendantes afin de résoudre tous les conflits. Nous supposerons donc que, par correction des données d'entrée, nous sommes toujours dans le cas où $t_{out}(e) - t_{in}(e) < l(e)$.

3.3.1 Influence d'un arc sur le nombre de valeurs en vie

Nous souhaitons dans cette section calculer formellement le nombre de valeurs intermédiaires correspondant à une dépendance particulière de notre graphe. Plus précisément, à un instant donné, pour une paire d'instructions dépendantes, nous souhaitons exprimer le nombre de valeurs créées et le nombre de valeurs consommées depuis le début de l'ordonnancement par les différentes instances. Nous souhaitons également en déduire le nombre minimal de valeurs encore en vie requis par ces deux instructions à cet instant.

Considérons un instant t de l'ordonnancement. Pour un arc donné $e = (u, v) \in E_{val}$, le nombre de valeurs créées par l'instruction source u depuis le début de l'ordonnancement jusqu'à t est égal au nombre d'itérations, donc de valeurs de i, pour lesquelles (u, i) a été ordonnancée avant $t - t_{out}(e)$. Donc, le nombre de valeurs produites pour e, $prod_r(e, t)$, à l'instant t et depuis le début de l'ordonnancement est :

$$prod_r(e, t) = |\{i \in \mathbb{N}^* \mid \sigma(u, i) + t_{out}(e) \leq t\}|$$

Si nous supposons que λ est strictement positif (ce qui est le cas de tout code non transformé par pipeline logiciel, ou encore de tout code produit par notre algorithme 6 de compaction de boucle[2]), comme $i \in \mathbb{N}^*$ et $\sigma(u, i)$ est une fonction de i croissante, alors $prod_r(e, t)$ est égal à la valeur maximale de i vérifiant :

$$\begin{array}{lrcl} & \sigma(u, i) + t_{out}(e) & \leq & t \\ \iff & a_u + \lambda(i + r(u)) + t_{out}(e) & \leq & t \\ \iff & i & \leq & \dfrac{t - a_u - t_{out}(e)}{\lambda} - r(u) \end{array}$$

Donc, puisque i est entier, nous obtenons :

$$prod_r(e, t) = \left\lfloor \frac{t - a_u - t_{out}(e)}{\lambda} \right\rfloor - r(u)$$

De la même façon, si nous cherchons à exprimer le nombre de valeurs intermédiaires associées à e, et consommées depuis le début de l'ordonnancement jusqu'à un instant t par les opérations $(v, i+d(e))$, dépendantes des opérations (u, i), nous obtenons :

$$\begin{array}{rcl} cons_r(e, t) & = & |\{i \in \mathbb{N}^* \mid \sigma(v, i + d(e)) + t_{in}(e) \leq t\}| \\ & = & \left\lfloor \dfrac{t - a_v - t_{in}(e)}{\lambda} \right\rfloor - d(e) - r(v) \end{array}$$

[2]De façon plus générale, $\lambda \geq 0$ est une supposition raisonnable dans la plupart des situations de pipeline logiciel.

Nous pouvons alors exprimer, à l'aide de $prod_r(e,t)$ et $cons_r(e,t)$, le nombre minimal de valeurs encore en vie requises pour l'arc e à un instant t :

$$\begin{aligned} vie_r(e,t) &= prod_r(e,t) - cons_r(e,t) \\ &= \left\lfloor \frac{t - a_u - t_{out}(e)}{\lambda} \right\rfloor - r(u) - \left\lfloor \frac{t - a_v - t_{in}(e)}{\lambda} \right\rfloor + d(e) + r(v) \\ &= d(e) + r(v) - r(u) + \left\lfloor \frac{t' - a_u - t_{out}(e)}{\lambda} \right\rfloor - \left\lfloor \frac{t' - a_v - t_{in}(e)}{\lambda} \right\rfloor \end{aligned}$$

où $t' = t \mod \lambda$. Nous définissons alors :

$$s(e,t) = \left\lfloor \frac{t' - a_u - t_{out}(e)}{\lambda} \right\rfloor - \left\lfloor \frac{t' - a_v - t_{in}(e)}{\lambda} \right\rfloor$$

Nous obtenons finalement :

$$vie_r(e,t) = d_r(e) + s(e,t)$$

Ceci n'étant valable que pour $e \in E_{val}$, pour tout arc $e \notin E_{val}$, nous posons $vie_r(e,t) = 0$. Et nous pouvons remarquer que $\forall k \in \mathbb{Z}$:

$$vie_r(e,t) = vie_r(e, t + \lambda k)$$

De façon logique, l'ordonnancement étant périodique, il en est de même du nombre de valeurs en vie (et la période est la même).

3.3.2 Influence d'un sommet sur le nombre de valeurs en vie

Supposons dans un premier temps que chaque sommet u ne produise qu'un seul résultat et donc qu'il existe une unique valeur k telle que, pour tout arc $e = (u,.) \in E_{val}$, on ait $t_{out}(e) = k$. Dans ce cas, à un instant donné t, pour tous ces arcs $e = (u,.) \in E_{val}$, $prod_r(e,t)$ est le même et correspond effectivement aux mêmes valeurs produites par l'instruction u. Il peut donc n'être stocké qu'une seule fois pour servir d'opérande à tous les sommets successeurs. Le nombre de valeurs à stocker va alors dépendre du dernier consommateur successeur de u dans le graphe. Autrement dit, les valeurs produites depuis le début de l'ordonnancement sont les mêmes, mais les sommets successeurs ne les consomment pas tous au même instant : l'espace de stockage utilisé pour une valeur ne pourra être libéré qu'une fois le dernier successeur exécuté. Le dernier successeur étant le sommet ayant consommé le moins de valeurs, le nombre de valeurs produites par un sommet et encore en vie à un instant t de l'ordonnancement est donc :

$$\begin{aligned} \forall u \in V_{val}, \forall e = (u,.) \in E_{val}, valeurs_r(u,t) &= prod_r(e,t) - \min_{e'=(u,.)\in E_{val}} cons_r(e',t) \\ &= \max_{e'=(u,.)\in E_{val}} vie_r(e',t) \end{aligned}$$

Notons que, par convention, pour tout sommet $u \notin V_{val}$ nous posons $valeurs_r(u,t) = 0$. Nous pouvons remarquer que comme $vie_r(e,t)$ est périodique de période λ, alors $\forall k \in \mathbb{Z}$:

$$valeurs_r(u,t) = valeurs_r(u, t + \lambda k)$$

Replaçons-nous alors dans le cas général où un sommet peut produire plusieurs résultats. Dans ce cas, pour chacun des résultats produits les remarques précédentes tiennent toujours : pour tous

les arcs e correspondant un résultat particulier, $prod_r(e,t)$ est le même et le nombre de valeurs à stocker pour ce résultat à un instant t est le maximum de $vie_r(e,t)$. Donc, si nous nous trouvons dans le cas d'un sommet u produisant plusieurs résultats (par exemple, une division entière produit un quotient et un reste), nous pouvons supposer que u est en fait composé de plusieurs sommets virtuels $\{u_1,\ldots,u_n\}$ ayant tous la même valeur de *retiming* et ayant chacun besoin de $valeurs_r(u_i,t)$ (où chaque u_i a pour arcs sortants une partie des arcs sortants de u) emplacements de stockage à tout instant t de l'ordonnancement. Par la suite, par souci de simplicité, nous supposerons que chaque sommet ne produit qu'un seul résultat. Tous nos résultats sont aisément transposables au cas de plusieurs résultats produits en réécrivant les contraintes de coût séparément pour chaque sommet virtuel (et en laissant les contraintes de *retiming* inchangées).

3.3.3 Quantité de mémoire requise

Chaque sommet u du graphe ayant besoin de $valeurs_r(u,t)$ espaces de stockage à un instant t de l'ordonnancement, nous en déduisons la quantité totale de mémoire requise à cet instant :

$$coût_r(t) = \sum_{u \in V_{val}} valeurs_r(u,t)$$

Finalement, la quantité de mémoire minimale nécessaire pour l'ensemble de la boucle, *maxlive*, sera déterminée par l'instant qui en requiert le plus. Et comme $valeurs_r(u,t)$ est périodique de période λ, nous obtenons :

$$maxlive_r = \max_{0 \le t < \lambda} coût_r(t)$$

Cependant, ce résultat n'est pas pleinement satisfaisant. En effet, il implique qu'un algorithme de minimisation de $maxlive_r$ soit au mieux pseudo-polynomial, car le nombre de valeurs du coût à considérer est λ, alors que λ est un nombre, donc généralement représenté par $\log(\lambda)$ chiffres. De plus, intuitivement, vérifier tous les instants semble superflu : pendant les instants de l'ordonnancement durant lesquels aucune opération ne crée de valeur et aucune n'en consomme, la quantité $maxlive_r$ ne varie pas. Nous pouvons même aller plus loin en disant que $maxlive_r$ n'augmente que lorsqu'une opération produit son résultat, soit à $|V|$ instants différents.

Proposition 6 *Soit un graphe de dépendance $G = (V, E_{val} \cup E_{dep})$, un ordonnancement cyclique $\sigma(u,i) = a_u + \lambda i$ de G, un* retiming *r de G et l'ensemble $dates_\sigma = \{a_u + t_{out}(e) \mod \lambda \mid e = (u,.) \in E_{val}\}$, alors :*

$$\max_{0 \le t < \lambda} coût_r(t) = \max_{t \in dates_\sigma} coût_r(t)$$

Preuve Soit un instant $0 \le t < \lambda$ et un arc $e = (u,v) \in E_{val}$, nous avons :

$$vie_r(e,t) = d_r(e) + \left\lfloor \frac{t - a_u - t_{out}(e)}{\lambda} \right\rfloor - \left\lfloor \frac{t - a_v - t_{in}(e)}{\lambda} \right\rfloor$$

Soit $a_u + t_{out}(e) = \alpha_e \lambda + \beta_e$ où $0 \le \beta_e < \lambda$ et $a_v + t_{in}(e) = \delta_e \lambda + \gamma_e$ où $0 \le \gamma_e < \lambda$ (division euclidienne).
Nous obtenons :

$$vie_r(e,t) = d_r(e) - \alpha_e + \delta_e + \left\lfloor \frac{t - \beta_e}{\lambda} \right\rfloor - \left\lfloor \frac{t - \gamma_e}{\lambda} \right\rfloor$$

avec $-\lambda < t - \beta_e < \lambda$ et $-\lambda < t - \gamma_e < \lambda$. Posons :

$$s'(e,t) = \left\lfloor \frac{t - \beta_e}{\lambda} \right\rfloor - \left\lfloor \frac{t - \gamma_e}{\lambda} \right\rfloor$$

Les valeurs β_e et γ_e calculées peuvent être respectivement interprétées comme les dates de production et de consommation d'une valeur associée à e dans un motif de taille λ. Soit t' tel que $0 \leq t' \leq t$ si $t < \beta_e$ et $\beta_e \leq t' \leq t$ sinon. Nous avons alors 4 possibilités illustrées par la figure 3.4 :

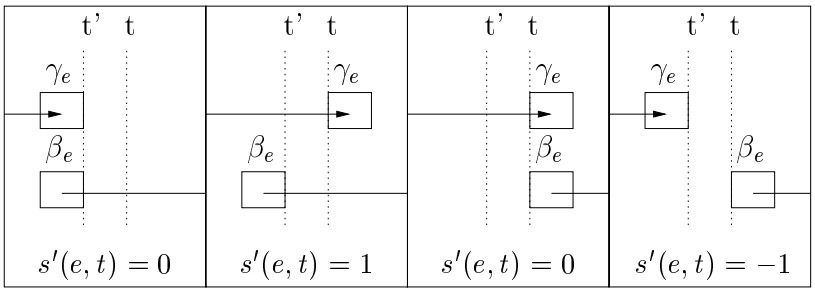


FIG. 3.4 – Valeurs des $s'(e,t)$ selon les positions relatives de β_e, γ_e et t.

1. $t < \beta_e$ et $t < \gamma_e$, alors $s'(e,t) = 0$ et $s'(e,t') = 0$;
2. $t < \beta_e$ et $t \geq \gamma_e$, alors $s'(e,t) = -1$ et soit $t' < \gamma_e$ et $s'(e,t') = 0$ soit $t' \geq \gamma_e$ et $s'(e,t') = -1$;
3. $t \geq \beta_e$ et $t < \gamma_e$, alors $s'(e,t) = 1$ et $s'(e,t') = 1$;
4. $t \geq \beta_e$ et $t \geq \gamma_e$, alors $s'(e,t) = 0$ et soit $t' < \gamma_e$ et $s'(e,t') = 1$ soit $t' \geq \gamma_e$ et $s'(e,t') = 0$;.

Dans tous les cas $s'(e,t') \geq s'(e,t)$.

Soit $t_{pred}(t) = \max_{e' \in \{e' \in E_{val} \mid \beta_{e'} \leq t\}} \beta_{e'}$, alors pour tout arc $e \in E_{val}$, $t_{pred}(t)$ vérifie la condition sur t' précédente, c'est-à-dire $0 \leq t_{pred}(t) \leq t$ si $t < \beta_e$ et $\beta_e \leq t_{pred}(t) \leq t$ sinon, donc :

$$\begin{aligned} \forall e \in E_{val}, s'(e, t_{pred}(t)) &\geq s'(e,t) \\ \forall e \in E_{val}, vie_r(e, t_{pred}(t)) &\geq vie_r(e,t) \\ \forall u \in V, valeurs_r(u, t_{pred}(t)) &\geq valeurs_r(u,t) \\ coût_r(t_{pred}(t)) &\geq coût_r(t) \end{aligned}$$

Comme seuls les t vérifiant $t \in dates_\sigma$ sont tels que $t_{pred}(t) = t$, ce sont les seuls utiles dans l'expression du max. □

Il en résulte que nous pouvons nous contenter de ne considérer que le coût aux instants $t \in dates_\sigma = \{a_u + t_{out}(e) \mod \lambda \mid e = (u,.) \in E_{val}\}$. Dans notre modèle simplifié, dans lequel chaque sommet ne produit qu'un seul résultat, nous obtenons au plus $\min(|V|, \lambda)$ instants différents (et $\min(|E_{val}|, \lambda)$ dans le cas général). Dans tous les cas, le nombre d'instants à considérer est polynomial en la taille de notre graphe.

3.4 Réordonnancement par étages

Comme nous l'avons vu dans la section 3.3 précédente, la quantité totale de mémoire nécessaire au stockage des valeurs intermédiaires d'une boucle, ou *maxlive*, dépend des valeurs $vie_r(e,t)$, valeurs qui à t fixé dépendent de r et des a_u, donc de σ. Rappelons que, comme nous l'avons déjà précisé dans l'introduction, *maxlive* peut être considérée comme une bonne mesure de la quantité de registres nécessaire à l'exécution d'une boucle une fois l'allocation faite (voir [37, 71, 95]). L'idée du réordonnancement par étages (ou *stage scheduling*) est d'utiliser un ordonnancement existant et de chercher à faire baisser *maxlive* en ne changeant que la partie *retiming* de l'ordonnancement. De

cette manière, le motif de l'ordonnancement reste inchangé, donc les contraintes de ressources sont naturellement respectées et la performance de l'ordonnancement reste la même (λ reste également inchangé). Ainsi, le problème s'en trouve radicalement simplifié : l'objectif n'est plus que de minimiser la quantité de mémoire telle que nous l'avons exprimée précédemment, et les contraintes ne sont plus que des contraintes de correction d'ordonnancement.

Définition 21 (*retiming* correct par rapport à un ordonnancement) *Soit un graphe de dépendance $G = (V, E)$, un ordonnancement cyclique $\sigma(v, i) = a_v + \lambda i$ et un* retiming *r de G. Nous définissons $\forall e = (u, v) \in E, d_{min}(e) = \left\lceil \frac{a_u - a_v + l(e)}{\lambda} \right\rceil$. Nous disons alors que r est correct par rapport à σ si et seulement si :*

$$\forall e \in E, d_r(e) \geq d_{min}(e)$$

Proposition 7 *Soit un graphe de dépendance $G = (V, E)$, un ordonnancement cyclique $\sigma(v, k) = a_v + \lambda k$ et un* retiming *r de G. Si r est correct par rapport à σ, alors $\sigma'(v, i) = \sigma(v, i) + \lambda r(v)$ est un ordonnancement vérifiant les contraintes de dépendance de G.*

Preuve Pour tout arc, nous avons :

$$\begin{array}{lrcl}
 & d_{min}(e) & \leq & d_r(e) \\
\iff & \left\lceil \frac{a_u - a_v + l(e)}{\lambda} \right\rceil & \leq & d(e) + r(v) - r(u) \\
\iff & a_u - a_v + l(e) & \leq & (i + d(e) - i + r(v) - r(u))\lambda \\
\iff & a_u + \lambda(i + r(u)) + l(e) & \leq & a_v + \lambda(i + d(e) + r(v)) \\
\iff & \sigma'(u, i) + l(e) & \leq & \sigma'(v, i + d(e))
\end{array}$$

Donc les contraintes de dépendance de G sont vérifiées par σ'. □

La proposition suivante prouve également que, pour tout ordonnancement correct σ, le nombre de valeurs en vie $vie_r(e, t)$ associé à un arc e à un instant t est toujours positif.

Proposition 8 *Soit un graphe de dépendance $G = (V, E)$, r un* retiming *de G et un ordonnancement cyclique $\sigma(v, k) = a_v + \lambda k$. Si r est un* retiming *correct par rapport à σ, alors :*

$$\forall e \in E_{val}, \forall t \in \mathbb{Z}, vie_r(e, t) \geq 0$$

Preuve Par hypothèse, pour tout arc $e \in E$, nous avons $t_{out}(e) - t_{in}(e) < l(e)$. De plus, comme r est correct par rapport à σ, $\sigma + \lambda r$ est un ordonnancement correct, donc :

$$\begin{array}{lrcl}
 & \sigma(u, i) + \lambda r(u) + t_{out}(e) & < & \sigma(v, i) + \lambda r(v) + t_{in}(e) \\
\iff & a_u + \lambda(i + r(u)) + t_{out}(e) & < & a_v + \lambda(i + d(e) + r(v)) + t_{in}(e) \\
\iff & t - a_v - t_{in}(e) & < & \lambda(d(e) + r(v) - r(u)) + t - a_u - t_{out}(e) \\
\iff & \frac{t - a_v - t_{in}(e)}{\lambda} & < & d(e) + r(v) - r(u) + \frac{t - a_u - t_{out}(e)}{\lambda} \\
\iff & \left\lfloor \frac{t - a_v - t_{in}(e)}{\lambda} \right\rfloor & \leq & d_r(e) + \left\lfloor \frac{t - a_u - t_{out}(e)}{\lambda} \right\rfloor \\
\iff & 0 & \leq & vie_r(e, t)
\end{array}$$

□

Évidemment, comme le réordonnancement par étages ne change que r, c'est-à-dire une partie seulement des paramètres, son efficacité sera moins grande que la résolution du problème d'ordonnancement de coût mémoire minimal sous contraintes de ressources. Cependant, si nous sommes en mesure de résoudre efficacement le problème, le résultat ne peut être que bénéfique puisque la performance de l'ordonnancement est conservée.

3.4.1 Le réordonnancement par étages est NP-complet au sens fort

Dans cette section, nous prouvons que le réordonnancement par étages est un problème NP-complet au sens fort. Commençons par poser le problème de décision associé :

Problème : RÉORDONNANCEMENT PAR ÉTAGES

Instance Un graphe de dépendance $G = (V, E_{val} \cup E_{dep})$ pondéré sur ses arcs par d, l, t_{in} et t_{out}, un ordonnancement cyclique $\sigma(v, k) = a_v + k\lambda$, et un entier naturel K.

Question Existe-t-il un *retiming* r de G, correct par rapport à σ, tel que $maxlive_r \leq K$?

Le preuve se fait par réduction polynomiale de 3SAT (problème LO2 de [47, p. 259]), dont nous rappelons ici la formulation :

Problème : 3SAT

Instance Un ensemble U de n variables booléennes et un ensemble C de m clauses sur U (pour chaque variable $u \in U$, u et $\bar{u}$ sont appelés des littéraux sur U, une clause est un ensemble de littéraux sur U) tel que pour toute clause $c \in C$, $|c| = 3$.

Question Existe-t-il un assignement de vérité pour U (c'est-à-dire une fonction $T : U \longrightarrow \{V, F\}$, qui s'étend aux littéraux de la façon suivante : $T(\bar{u}) = V$ si $T(u) = F$ et $T(\bar{u}) = F$ si $T(u) = V$) tel que toute clause $c \in C$ contienne au moins un littéral vrai ($\exists x \in c, T(x) = V$) ?

Nous sommes alors prêts à prouver le théorème suivant établissant la NP-complétude de RÉORDONNANCEMENT PAR ÉTAGES.

Théorème 2 *Le problème* RÉORDONNANCEMENT PAR ÉTAGES *est NP-complet au sens fort.*

Preuve Nous commençons par prouver que le problème appartient à NP, puis nous décrivons notre transformation polynomiale d'une instance de 3SAT en instance de RÉORDONNANCEMENT PAR ÉTAGES, enfin nous montrons l'équivalence entre instances positives.

Le problème est dans NP Considérons une instance positive de notre problème : un graphe $G = (V, E)$ pondéré sur ses arcs par d, l, t_{in} et t_{out}) où $E = E_{val} \cup E_{dep}$, un ordonnancement cyclique $\sigma(v, k) = a_v + k\lambda$ et un entier naturel K. Soit r une solution à cette instance. Nous allons prouver qu'il existe une autre solution r' à notre instance dont les valeurs sont bornées par une fonction polynomiale des données du problème.

Pour cela, nous construisons $G' = (V, E' = E_{val} \cup E'_{other}, d')$ à partir de G et r de la façon suivante :

- nous partons de $G' = G$ et $d' = d$;
- pour tout arc $e = (u, v) \in E_{val}$, nous créons deux arcs supplémentaires dans E'_{other}, l'un e_1 de u vers v de poids $d'(e_1) = r(u) - r(v)$ et l'autre e_2 de v vers u de poids $d'(e_2) = r(v) - r(u)$. Nous posons $d_{min}(e_1) = d_{min}(e_2) = 0$ (notons que nous conservons l'arc e dans G').

Par construction de G', r vérifie $d'_r(e) = 0$ pour tous les nouveaux arcs (les arcs tels que $e \in E'_{other}$ mais $e \notin E_{dep}$). Donc pour ces arcs $d'_r(e) \geq d_{min}(e)$. Comme r est un *retiming* correct par rapport à σ, nous en déduisons que pour tous les autres arcs $e \in E$ nous avons $d_r(e) \geq d_{min}(e)$. Donc le graphe contraint par d_{min} de G'_r, ou $G'_r - d_{min}$, ne contient aucun arc de poids négatif, donc aucun circuit de poids négatif. Comme le *retiming* conserve le poids d'un circuit (corollaire 1), nous en déduisons que $G' - d_{min}$ ne contient aucun circuit de poids négatif. Donc, en utilisant l'algorithme 2, nous pouvons trouver un *retiming* r' de G' tel que pour tout arc $e \in E', d'_{r'}(e) \geq d_{min}(e)$. Ce *retiming* est donc un *retiming* correct par rapport à σ. De plus, par construction de G', pour tout arc $e \in E_{val}$ nous avons $d'(e_1) + r'(v) - r'(u) \geq 0$ et $d'(e_2) + r'(u) - r'(v) \geq 0$, donc $r'(v) - r'(u) \geq r(v) - r(u)$ et $r'(u) - r'(v) \geq -(r(v) - r(u))$, donc $r'(v) - r'(u) = r(v) - r(u)$. Il s'ensuit que $d_{r'}(e) = d_r(e)$ et que le coût de G_r est le même que celui de $G_{r'}$. Donc r' est une solution à notre instance.

Par ailleurs, comme r' est déterminé par l'algorithme 2, pour tout sommet $v \in V$, $r'(v)$ est positif et égal à l'opposé du poids d'un plus court chemin d'un sommet $u \in V$ vers v dans $G' - d_{min}$. Donc $r'(v) \leq \sum_{e \in E'} |d'(e) - d_{min}(e)| \leq \sum_{e \in E'} |d'(e)|$. Il nous reste à borner d'.

Pour tous les arcs $e \in E, d'(e) = d(e)$, et pour chaque arc $e \in E_{val}$, il y a deux nouveaux arcs de poids respectifs $r(v) - r(u)$ et $-(r(v) - r(u))$. Comme r est correct par rapport à σ, nous avons :

$$\begin{array}{rrcl} & d_r(e) & \geq & d_{min}(e) \\ \iff & r(v) - r(u) & \geq & d_{min}(e) - d(e) \\ \iff & r(v) - r(u) & \geq & -d(e) \end{array}$$

et comme r est solution, pour tout $0 \leq t < \lambda$ nous avons :

$$\begin{array}{rrcl} & vie_r(e,t) & \leq & K \\ \iff & d_r(e) + s(e,t) & \leq & K \\ \iff & r(v) - r(u) & \leq & K - d(e) - s(e,t) \end{array}$$

Donc, en regroupant tout, nous obtenons pour tout arc $e \in E'$:

$$|d'(e)| \leq \max(|d(e)|, \max_{0 \leq t < \lambda} |K - d(e) - s(e,t)|)$$

Donc pour tout sommet $v \in V$, $r'(v)$ est borné par une fonction polynomiale de notre instance, donc r' est un certificat polynomial de notre instance et RÉORDONNANCEMENT PAR ÉTAGES est dans NP.

Transformation Soit une instance (U, C) de 3SAT (comprenant n variables et m clauses), nous définissons $f(U, C) = (G, \sigma, K)$ comme une transformation polynomiale de (U, C) en instance de RÉORDONNANCEMENT PAR ÉTAGES de la façon suivante :

- nous partons d'un graphe G composé de trois sommets s, p_0 et p avec $a_s = 0$, $a_{p_0} = 2m + 1$ et $a_p = 2m + 2$. Tous les arcs que nous créerons par la suite appartiendront à E_{val}. Dans cette preuve, par souci de simplicité, nous désignons E_{val} par E, puisque E_{dep} est vide ;
- nous considérons l'ensemble des clauses $C = \{c_1, \ldots, c_m\}$ comme un ensemble ordonné (avec un ordre quelconque), nous avons donc $c_1 < c_2 < \ldots < c_m$. Pour chaque clause $c \in C$, avec i tel que $c = c_i$, et pour chaque littéral $l \in c$, il existe une variable $x \in U$ telle que l soit un littéral construit à partir de x (c'est-à-dire tel que $l = x$ ou $l = \bar{x}$). Nous créons 4 sommets nommés respectivement c_x, c^0_x, c^v_x et c^c_x et 4 arcs respectivement entre les couples de sommets (c_x, c^v_x), (c^v_x, c_x), (c^0_x, c^c_x) et (c^c_x, c^0_x) et tous de poids 1. De plus, l est soit :

– un littéral positif, c'est-à-dire $l = x$. Dans ce cas, nous complétons notre construction pour créer la structure représentée par la figure 3.5 , en ajoutant un arc de c_x vers c_x^0 de poids 1 (les arcs sont annotés par leur poids et par une valeur entre parenthèses dont nous expliquerons la signification ultérieurement). Nous posons alors $a_{c_x} = a_{c_x^v} = 2(i-1)+1$, et $a_{c_x^0} = a_{c_x^c} = 2(i-1)+2$.

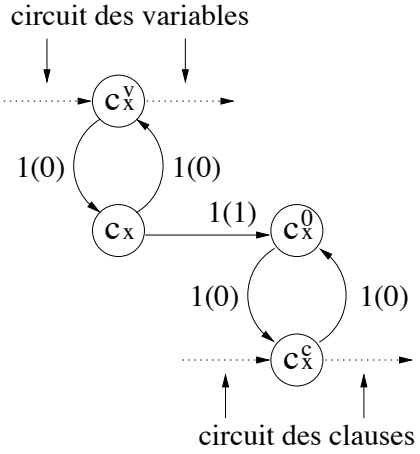


FIG. 3.5 – Un littéral positif.

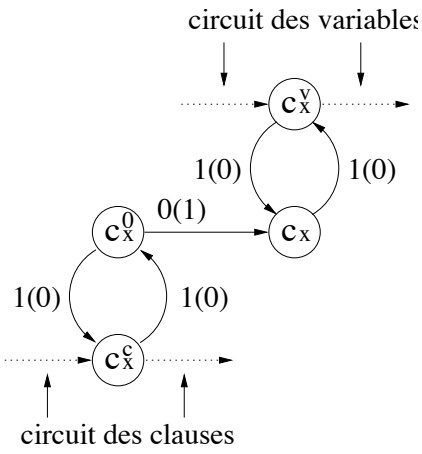


FIG. 3.6 – Un littéral négatif.

– un littéral négatif, c'est-à-dire $l = \bar{x}$. Dans ce cas, nous complétons notre construction pour créer la structure représentée par la figure 3.6, en ajoutant un arc de c_x^0 vers c_x de poids 0. Nous posons alors $a_{c_x^0} = a_{c_x^c} = 2(i-1)+1$, et $a_{c_x} = a_{c_x^v} = 2(i-1)+2$;

Nous verrons par la suite que selon la valeur de *retiming* de c_x, le nombre de valeurs en vie associé à ces deux structures va varier de manière opposée selon si le littéral est positif ou négatif. C'est cette différence de comportement des littéraux positifs et négatifs qui va nous permettre de mettre en relation le nombre de valeurs en vie dans nos structures et la valeur de vérité des variables. Notons que, sur les figures, les sommets sont ordonnés de gauche à droite par a_u croissants. Ceci devrait faciliter la compréhension par la suite (en particulier lors du calcul des $s(e,t)$).

– soit c_1 la première clause. Pour chaque littéral $l \in c_1$, construit à partir d'une variable x, nous ajoutons au graphe un arc de s vers $c_1{}_x^c$ de poids 0. Soit $c_m = \{l_1, l_2, l_3\}$ la dernière clause, avec l_1, l_2 et l_3 littéraux respectivement construits à partir des variables x, y et z. Nous ajoutons au graphe deux arcs respectivement de $c_m{}_y^c$ et $c_m{}_z^c$ vers p_0 et deux arcs respectivement de $c_m{}_x^c$ et p_0 vers p, tous de poids 0. Pour toute paire de clauses successives (c_i, c_{i+1}), telles que $c_i = \{l_1, l_2, l_3\}$ et $c_{i+1} = \{l'_1, l'_2, l'_3\}$, avec $l_1, l_2, l_3, l'_1, l'_2$ et l'_3 littéraux respectivement construits à partir des variables x, y, z, x', y' et z', nous ajoutons trois arcs respectivement entre les paires de sommets $(c_i{}_x^c, c_i{}_{x'}^c)$, $(c_i{}_y^c, c_i{}_{y'}^c)$ et $(c_i{}_z^c, c_i{}_{z'}^c)$ tous de poids 0. Nous ajoutons également au graphe un arc de p vers s de poids 1.

L'ensemble de ces arcs est représenté sur la figure 3.7. Nous pouvons remarquer que pour toute clause $c \in C$ et tout littéral $l \in c$ construit à partir de la variable x, c_x^c appartient à un circuit contenant s et p dont tous les arcs sont nuls sauf l'arc de p vers s. Cette construction cyclique sur les clauses va nous permettre d'imposer la même valeur de *retiming* à certains sommets appartenant à des clauses distinctes.

– pour chaque variable $x \in U$, il existe un sous-ensemble de C ordonné $C_x = \{c_1, \ldots, c_k\}$ tel que toute clause $c \in C_x$ contienne soit x soit $\bar{x}$, et donc un sommet c_x dans la structure associée

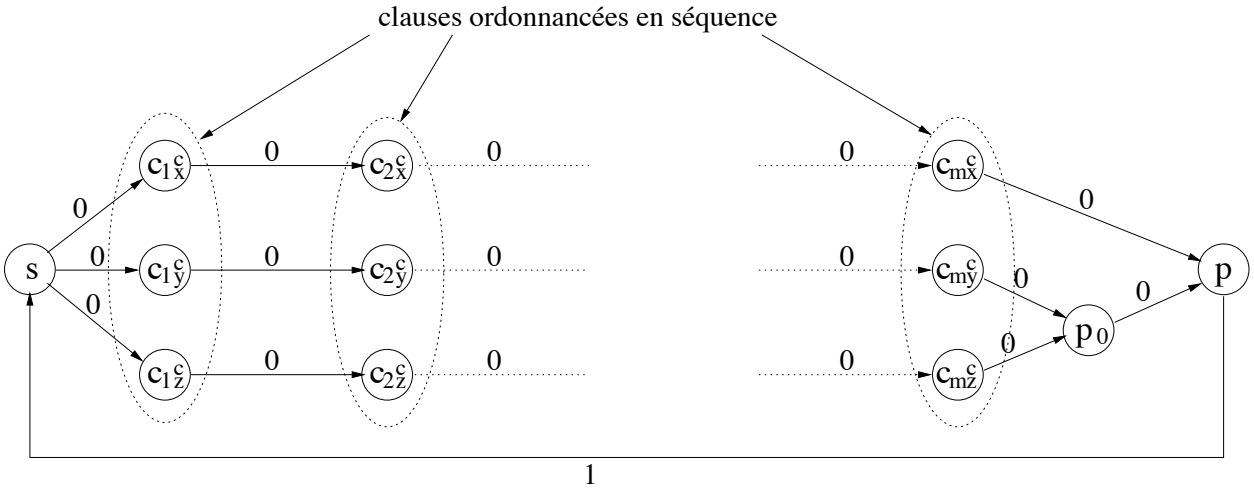


FIG. 3.7 – Multi-circuit reliant les clauses.

au littéral correspondant. Ces sommets forment un ensemble ordonné $\{c_{1x}, \ldots, c_{kx}\}$ (induit par l'ordre sur les clauses) dans lequel nous relions deux sommets successifs c_{ix} et c_{i+1x} par un arc de poids nul. Nous relions également le dernier sommet c_{kx} au premier c_{1x} par un arc de poids 1.

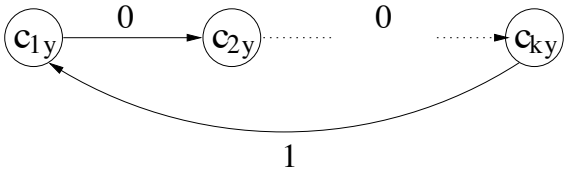


FIG. 3.8 – Circuit reliant les variables.

L'ensemble de ces arcs est représenté sur la figure 3.8. Nous pouvons remarquer que pour toute variable $x \in U$ tous les sommets associés à cette variable font partie d'un circuit dont tous les arcs sont nuls excepté celui reliant c_{kx} à c_{1x}. Ces circuits sur les variables vont nous permettre d'imposer la même valeur de *retiming* à tous les sommets correspondant à la même variable.

– pour tout arc $e \in E$ du graphe créé, nous posons $l(e) = t_{in}(e) = t_{out}(e) = 1$. Nous définissons $\lambda = 2m + 4$ et pour tout sommet $u \in V$, $\sigma(u, i) = a_u + \lambda i$. Enfin $K = 12(m - 1) + 14$.

Pour chaque littéral, nous avons 4 sommets et 5 arcs, donc $12m$ sommets et $15m$ arcs pour l'ensemble des littéraux. Pour le multi-circuit reliant les clauses, nous avons besoin d'un arc entrant par littéral, plus les sommets s, p_0 et p et leurs arcs entrants (5 en tout), donc 3 sommets et $3m+5$ arcs pour le multi-circuit. Enfin, pour chacun des circuits reliant les variables nous avons besoin au pire de m arcs si la variable apparaît dans toutes les clauses, donc au pire nm pour l'ensemble de ces circuits. Au total notre transformation contient donc au pire $12m+3$ sommets et $nm+18m+5$ arcs, donc elle est polynomiale en la taille de (U, C).

Remarque sur les $s(e,t)$: Nous pouvons remarquer que pour tout sommet $u \in V$, $0 \leq a_u + t_{in}(e) < 2m + 4 \leq \lambda$ et $0 \leq a_u + t_{out}(e) < 2m + 4 \leq \lambda$, nous nous trouvons donc dans une situation similaire à celle que nous avions dans la proposition 6 après division de ces quantités par λ. Autrement dit, pour tout arc $e = (u, v) \in E$ et tout $0 \leq t < \lambda$, nous pouvons calculer $s(e, t)$ de

la même façon que le $s'(e,t)$ de la proposition 6, et nous en déduisons que $s(e,t) = -1, 0$ ou 1 selon le cas. De plus, comme $t_{in}(e) = t_{out}(e) = 1$, nous obtenons finalement un calcul simplifié des $s(e,t)$ pour $0 \leq t < \lambda$ (voir figure 3.9 pour une illustration des 4 cas de figure) :

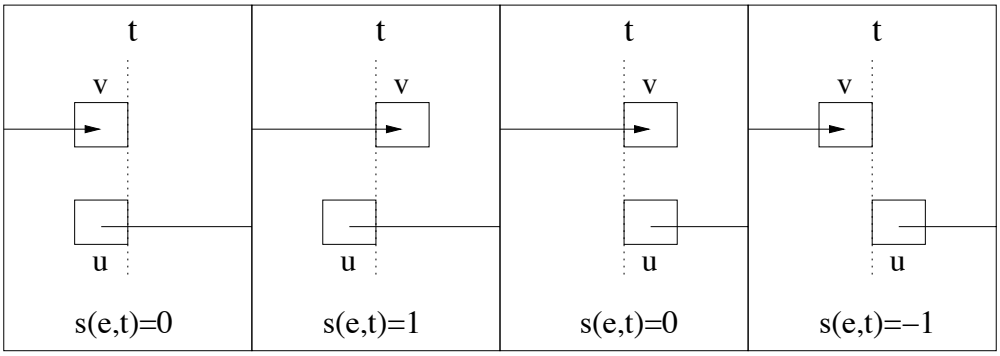


FIG. 3.9 – Illustration pour les valeurs de $s(e,t)$.

1. si $t \leq a_u$ et $t \leq a_v$ alors $s(e,t) = 0$;
2. si $t \leq a_u$ et $t > a_v$ alors $s(e,t) = -1$;
3. si $t > a_u$ et $t \leq a_v$ alors $s(e,t) = 1$;
4. si $t > a_u$ et $t > a_v$ alors $s(e,t) = 0$.

Remarque sur les $s(e,t)$ d'une clause : Toute clause $c_i \in \{c_1, \ldots, c_m\}$ est composée de trois littéraux chacun associé à une structure de graphe dont les sommets sont ordonnancés aux temps $2(i-1)+1$ et $2(i-1)+2$ de chaque itération. Donc, pour toute paire de sommets (u,v) d'une des structures de la clause correspondant à un arc e, si $t \leq 2(i-1)+1$ nous sommes toujours dans le cas 1 de la remarque précédente, et si $t > 2(i-1)+2$ nous sommes toujours dans le cas 4. Dans les deux cas, $s(e,t) = 0$. Donc par la suite, pour chaque clause c_i, nous ferons la distinction entre l'instant $t = 2(i-1)+2$ que nous appellerons instant local et les autres instants que nous appellerons instants non locaux ou externes. Sur les figures 3.5 et 3.6, les arcs d'une clause sont annotés par leur poids et par une valeur entre parenthèse. Cette valeur correspond au $s(e,t)$ de l'arc à l'instant local de la clause. Le $s(e,t)$ aux autres instants étant toujours nul, nous n'avons pas jugé utile de le préciser.

Remarque sur les $d_{min}(e)$: Le $d_{min}(e)$ de tout arc $e = (u,v)$ de notre graphe est très simple à calculer, en effet, dans notre transformation, nous avons choisi $l(e) = 1$, donc $0 \leq a_u + l(e) < \lambda$. Donc $-\lambda < a_u - a_v + 1 < \lambda$ et

$$d_{min}(e) = \begin{cases} 0 \text{ si } a_u < a_v \\ 1 \text{ sinon} \end{cases}$$

Réduction Soit (U,C) une instance de 3SAT et $(G,\sigma,K) = f(U,C)$. Nous devons prouver que :
- si (U,C) est une instance positive de 3SAT, alors (G,σ,K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES : soit T un assignement de vérité pour U satisfaisant

toutes les clauses de C. Nous posons :

$$\begin{array}{l} r(s) = r(p_0) = r(p) = 0 \\ \forall c \in C, \quad \forall x \in \{x \in U \mid x \in c \text{ ou } \bar{x} \in c\}, \\ \qquad\qquad r(c_x) = r(c_x^v) = \left\{ \begin{array}{l} 1 \text{ si } T(x) = V \\ 0 \text{ sinon} \end{array} \right. \\ \qquad\qquad r(c_x^0) = r(c_x^c) = 0 \end{array}$$

Lorsque, pour une variable $x \in U, T(x) = F$, toutes les structures associées aux littéraux x et

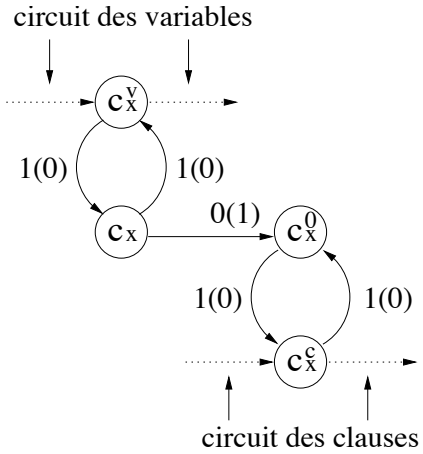


FIG. 3.10 – Un littéral positif après *retiming*.

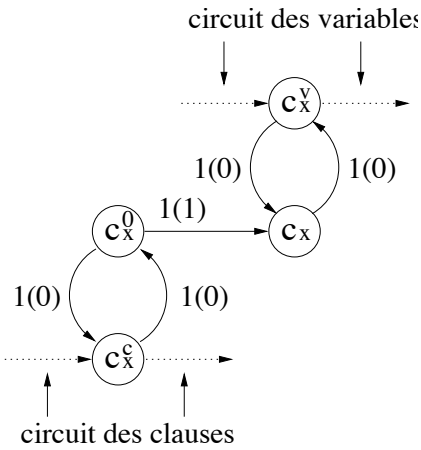


FIG. 3.11 – Un littéral négatif après *retiming*.

$\bar{x}$ ne subissent aucun *retiming* de leur sommet, le poids de leurs arcs reste donc celui indiqué par les figures 3.5 et 3.6. En revanche, lorsque $T(x) = V$, toutes les structures associées aux littéraux x et $\bar{x}$ subissent un *retiming* dont la répercussion sur les poids est indiquée sur les figures 3.10 et 3.11. Nous allons alors examiner le nombre de valeurs en vie pour tous les sommets des structures que nous avons construites, afin de déterminer le coût total de notre *retiming*. Nous avons besoin d'examiner le coût pour les instants $0 \leq t < \lambda = 2m + 4$.

Considérons tout d'abord les sommets de la forme c_x^c. Ces sommets appartiennent au multi-circuit des clauses (figure 3.7) pour lequel tous les sommets ont une valeurs de *retiming* nulle, un de leurs arcs sortants appartient donc à ce circuit et a un poids nul (un des arcs en pointillés sur les figures 3.5 et 3.6). Leur deuxième arc sortant appartient à la structure de littéral dans laquelle ils ont été créés (figures 3.5 et 3.6). Il a un poids de 1 car la valeur de *retiming* de c_x^0 est nulle. En outre, ses valeurs de $s(e,t)$ sont nulles à tout instant t. Donc le coût du deuxième arc est toujours le maximum sur les arcs sortants de c_x^c, donc le coût de c_x^c est 1 à tout instant (variable écrite par c_x^0 et lue par c_x^c à l'itération suivante).

De façon analogue, les sommets de la forme c_x^v appartiennent à un des circuits des variables, pour lequel tous les sommets ont la même valeur de *retiming* (0 ou 1 selon T). Un de leurs arcs sortants appartient donc à ce circuit. Il a soit un poids nul et un $s(e,t)$ qui nous importe peu, soit un poids de 1 et un $s(e,t)$ inférieur ou égal à 0 à tout instant (si c'est l'arc de retour). Leur deuxième arc sortant appartient à la structure de littéral dans laquelle ils ont été créés, il a un poids de 1 car la valeur de *retiming* de c_x est la même (par définition), et ses valeurs de $s(e,t)$ sont nulles à tout instant t. Donc le coût du deuxième arc est toujours le maximum sur les arcs sortants de c_x^v, donc le coût de c_x^v est 1 à tout instant.

Calculons alors le coût des sommets de la forme c_x et c_x^0. Leurs seuls arcs sortants sont ceux de la structure de littéral à laquelle ils appartiennent. Ces deux sommets sont reliés par un arc e et, selon si le littéral qui leur est associé dans c est positif ou négatif, l'un ou l'autre est le puits de cet arc. Le puits de l'arc e n'a qu'un seul arc sortant de poids 1 (par définition du *retiming*) dont les valeurs de $s(e,t)$ sont toujours nulles, son coût est donc de 1. La source de l'arc e possède toujours, en plus de e, un arc sortant de poids 1 dont les $s(e,t)$ sont toujours nuls (selon le cas soit vers c_x^c, soit vers c_x^v). Pour son coût, quatre cas se présentent :
- x apparaît tel quel dans c et $T(x) = V$, dans ce cas la source de e est le sommet c_x et son puits le sommet c_x^0. Comme $r(c_x) = 1$ (figure 3.10), le coût de c_x est alors déterminé par son arc sortant vers c_x^v, c'est-à-dire 1 à tout instant ;
- x apparaît tel quel dans c et $T(x) = F$, dans ce cas la source de e est le sommet c_x et son puits le sommet c_x^0. Comme $r(c_x) = 0$ (figure 3.5), le coût de c_x est alors déterminé par e, c'est-à-dire 2 à l'instant local et 1 aux autres instants ;
- x apparaît en tant que $\bar{x}$ dans c et $T(x) = V$, dans ce cas la source de e est le sommet c_x^0 et son puits le sommet c_x. Comme $r(c_x) = 1$ (figure 3.11), le coût de c_x^0 est alors déterminé par e, c'est-à-dire 2 à l'instant local et 1 aux autres instants ;
- x apparaît en tant que $\bar{x}$ dans c et $T(x) = F$, dans ce cas la source de e est le sommet c_x^0 et son puits le sommet c_x. Comme $r(c_x) = 0$ (figure 3.6), le coût de c_x^0 est alors déterminé par son arc sortant vers c_x^c, c'est-à-dire 1 à tout instant.

En regroupant ces cas deux-à-deux, nous obtenons deux cas pour la somme des coûts des sommets c_x et c_x^0 :
- si l est satisfait, donc $l = x$ et $T(x) = V$ ou bien $l = \bar{x}$ et $T(x) = F$, la somme des coûts de c_x et c_x^0 est 2 à tout instant.
- si l n'est pas satisfait, donc $l = x$ et $T(x) = F$ ou bien $l = \bar{x}$ et $T(x) = V$, la somme des coûts de c_x et c_x^0 est 3 à l'instant local et 2 aux autres instants.

Comme tous les littéraux d'une clause sont ordonnancés en parallèle (aux deux mêmes instants), leurs instants locaux sont les mêmes. Ainsi, si nous faisons la somme des coûts de tous les sommets construits pour une clause (les c_x, c_x^0, c_x^v et c_x^c de chaque littéral), nous obtenons un coût de 12 aux instants non locaux (car le coût de chaque sommet est 1 dans ce cas), et 12, 13, 14 ou 15 à l'instant local de la clause (selon le nombre de littéraux satisfaits, respectivement 3, 2, 1 ou aucun). De plus, nous pouvons vérifier que le coût de s est 1 à l'instant 1 et 0 aux autres, le coût de p_0 est 1 à l'instant $2m+2$ et 0 aux autres et le coût de p est 1 aux instants 0 et $2m+3$, et nul aux autres instants. Notons que les instants 0, 1, $2m+2$ et $2m+3$ ne sont locaux à aucune clause.

Finalement, en faisant la somme de tous les coûts, nous obtenons :
- à un instant qui n'est local à aucune clause, un coût total de $12m$ pour les clauses plus 0 ou 1 selon les coûts de s, p_0 et p ;
- à un instant local à une clause, qui par construction n'est local qu'à une seule clause, un coût total de $12(m-1)$ pour les clauses non locales et au plus 14 pour la clause locale puisque par définition de T au moins un littéral est satisfait dans chaque clause.

Le coût total à chaque instant est donc inférieur ou égal à K et (G, σ, K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES.

- si (G, σ, K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES, alors (U, C) est une instance positive de 3SAT. Soit r un *retiming* de G, correct par rapport à σ, tel que $maxlive_r \leq K$ dans G_r. Nous allons commencer par prouver une certain nombre de propriétés de r dues à la structure de G.

Tous les sommets c_x^c, par construction du circuit reliant les clauses, font partie d'un circuit contenant s et p dont tous les arcs sont nuls excepté l'arc de p vers s. De plus, pour tout arc $e = (u, v) \in E$ appartenant à un tel circuit, $d(e) = d_{min}(e)$ (la vérification est immédiate en reprenant la remarque sur les d_{min} de la page 89). Comme r est correct par rapport à σ, c'est-à-dire $d_r(e) \geq d_{min}(e)$, nous avons donc $r(u) \leq r(v)$. Et comme l'inégalité $r(u) \leq r(v)$ est vraie pour tous les arcs du circuit, c'est en fait une égalité, autrement dit $r(u) = r(v)$. Donc tous les sommets c_x^c ont la même valeur de *retiming*, égale à celle de s par exemple. Sans perte de généralité, nous supposons cette valeur nulle (autrement nous ajoutons une constante appropriée à toutes les valeurs de *retiming*).

Le même raisonnement sur les circuits des variables nous conduit à dire que pour chaque variable $x \in U$, tous les c_x^v ont la même valeur de *retiming*. De façon analogue, pour toute clause $c \in C$ contenant x ou $\bar{x}$, $r(c_x^0) = r(c_x^c) = 0$ et $r(c_x) = r(c_x^v)$, à cause des deux circuits présents dans toute structure associée à un littéral.

Pour toute variable $x \in U$, nous pouvons supposer que x apparaît dans au moins une clause $c_p \in C$ en tant que x et dans au moins une clause $c_n \in C$ en tant que $\bar{x}$. Dans le cas contraire, il suffit de fixer $T(x)$ à la bonne valeur (selon si x apparaît toujours tel quel ou en tant que $\bar{x}$) pour pouvoir satisfaire toutes les clauses le contenant. Donc, dans c_p, à cause de l'arc de poids 1 de c_{p_x} vers ${c_p}_x^0$, comme r est correct par rapport à σ, nous avons $r({c_p}_x^0) - r(c_{p_x}) + 1 \geq 0$, c'est-à-dire $r(c_{p_x}) \leq 1$. De même, dans c_n, à cause de l'arc de poids 0 de ${c_n}_x^0$ vers c_{nx}, nous avons $r(c_{nx}) - r({c_n}_x^0) \geq 0$, c'est-à-dire $r(c_{nx}) \geq 0$. Donc, comme tous les c_x ont la même valeur de *retiming*, pour toute variable $x \in U$, $0 \leq r(c_x) \leq 1$.

Nous définissons alors T, assignement de vérité sur U de la façon suivante :

$$\forall x \in U, T(x) = \begin{cases} V \text{ si } \exists c \in C, r(c_x) = 1 \\ F \text{ sinon} \end{cases}$$

Nous nous retrouvons donc dans une configuration dans laquelle les valeurs de *retiming* sont analogues à celles que nous avions construites à partir d'un assignement de vérité. Le même raisonnement sur les arcs et les sommets nous permet donc de dire que pour toute clause $c \in C$, nous obtenons un coût de 12 aux instants non locaux (car le coût de chaque sommet est 1 dans ce cas), et 12, 13, 14 ou 15 à l'instant local de la clause (selon le nombre de littéraux satisfaits, respectivement 3, 2, 1 ou aucun). Sachant que r est une solution de notre instance, le coût local de toute clause est donc inférieur ou égal à 14 (sinon, ajouté au coût externe de 12 des autres clauses, le coût total dépasserait $K = 12(m-1) + 14$). Donc, pour toute clause $c \in C$, il existe un littéral $l \in c$ tel que soit $l = x$ et $r(c_x) = 1$ soit $l = \bar{x}$ et $r(c_x) = 0$. Dans le premier cas cela signifie que $T(x) = V$, et dans le second cas, comme les $r(c_x)$ ont la même valeur dans toutes les clauses, cela signifie que $T(x) = F$. Dans les deux cas le littéral est vrai et la clause est satisfaite, donc (U, C) est une instance positive de 3SAT.

Ceci termine la preuve : RÉORDONNANCEMENT PAR ÉTAGES est NP-complet au sens fort. □

Notons que dans cette preuve, à cause de l'instance de RÉORDONNANCEMENT PAR ÉTAGES que nous construisons, nous pouvons dire que le problème reste NP-complet au sens fort même lorsque nous :

- restreignons toutes les latences 1 (ce qui montre aussi que dans notre modèle non pipeliné, où la contrainte sur les latences est qu'elles doivent toutes être égales au temps d'exécution du sommet source, le problème est toujours NP-complet) ;

- restreignons les t_{in} et t_{out} à 1 (donc même dans un modèle d'architecture simplifiée où tous les t_{in} sont égaux et tous les t_{out} également, le problème reste NP-complet) ;
- restreignons le degré entrant maximal d'une opération à 2 (donc même pour un code réaliste où chaque opération accepte au plus deux opérandes le RÉORDONNANCEMENT PAR ÉTAGES est NP-complet).

3.4.2 Restriction aux graphes acycliques (DAG)

Dans cette section, nous prouvons que le problème du réordonnancement par étage restreint aux graphes acycliques est NP-complet au sens fort. Ceci se fait par une légère modification de la preuve pour le cas général. Nous avons choisi de séparer le cas acyclique par souci de clarté, car la modification rend la preuve plus complexe. De plus, dans le cas acyclique nous ne prouvons pas que le problème reste NP-complet lorsque le degré entrant d'une opération est au plus 2. Notons que cette preuve ne va pas à l'encontre du résultat d'Eichenberger, Davidson et Abraham, qui prouvent dans [36] que le problème est polynomial pour un graphe sans cycles. En effet par acyclique nous entendons ici sans circuits mais le graphe peut très bien comporter des cycles (rappelons qu'un cycle est un circuit dans lequel les arcs peuvent être suivis dans les deux sens). Mais commençons par poser le problème de décision qui nous intéresse :

Problème : RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE

Instance Un graphe de dépendance acyclique $G = (V, E_{val} \cup E_{dep})$ pondéré sur ses arcs par d, l, t_{in} et t_{out}, un ordonnancement cyclique $\sigma(v, k) = a_v + k\lambda$, et un entier naturel K.

Question Existe-t-il un *retiming* r de G, correct par rapport à σ, tel que $maxlive_r \leq K$?

Nous prouvons alors que RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE est NP-complet au sens fort, en montrant que 3SAT se réduit polynomialement à ce problème (nous ne rappelons pas la formulation de 3SAT, que nous avons déjà rappelée à l'occasion de la preuve pour le cas général) :

Théorème 3 *Le problème* RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE *est NP-complet au sens fort.*

Preuve La preuve est extrêmement proche de celle du cas général et nous n'indiquerons ici que les modifications à apporter pour le cas acyclique.

Le problème est dans NP La preuve est la même que dans le cas général.

Transformation Soit une instance (U, C) de 3SAT. Nous définissons $f(U, C) = (G, \sigma, K)$ comme une transformation polynomiale de (U, C) en instance de RÉORDONNANCEMENT PAR ÉTAGES de la façon suivante (comme précédemment, nous désignerons E_{val} par E) :

- pour chaque clause $c \in C$, et $l \in c$, tel que l soit un littéral construit à partir de $x \in U$, nous construisons une sous-structure de celle construite dans le cas cyclique (tout y est identique, sauf certains éléments que nous enlevons). Deux cas se présentent, l est soit :
  - un littéral positif, c'est-à-dire $l = x$. Dans ce cas, nous ne construisons ni le sommet c_x^c (ni les arcs qui le contiennent), ni l'arc de c_x^v vers c_x. Nous obtenons la structure représentée par la figure 3.12 ;
  - un littéral négatif, c'est-à-dire $l = \bar{x}$. Dans ce cas, nous ne construisons ni le sommet c_x^v (ni les arcs qui le contiennent), ni l'arc de c_x^c vers c_x^0. Nous obtenons la structure représentée par la figure 3.13.

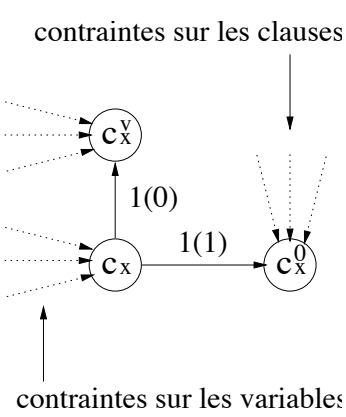


FIG. 3.12 – Un littéral positif (cas acyclique).

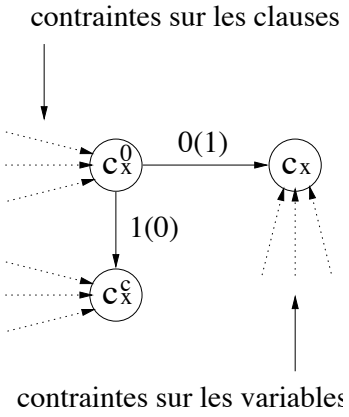


FIG. 3.13 – Un littéral négatif (cas acyclique).

– à la place du circuit construit pour relier l'ensemble des clauses, nous créons 3 sommets s^1, s^2 et s^3, tels que $a_{s^1} = a_{s^2} = a_{s^3} = 0$ et un sommet p tel que $a_p = 2m + 1$. Nous relions par un arc de poids nul chacun des sommets s^1, s^2 et s^3 à p ainsi qu'à chacun des sommets c_x^0 et, le cas échéant, c_x^c. Nous obtenons la structure représentée par la figure 3.14.

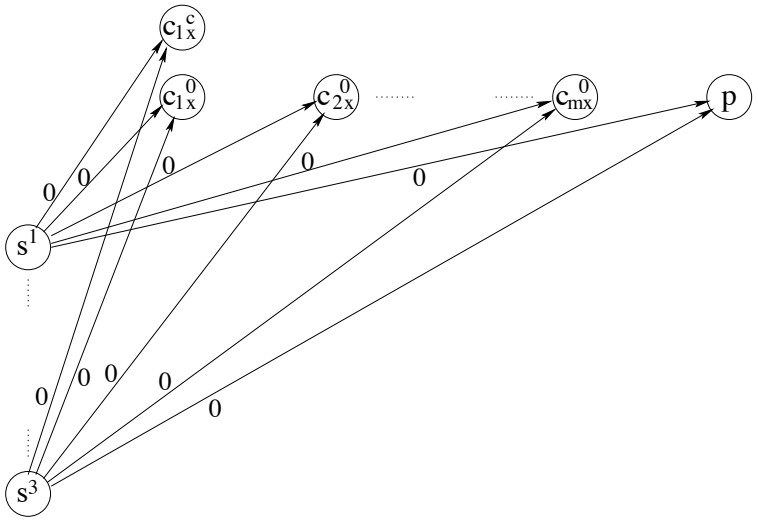


FIG. 3.14 – Structure remplaçant le multi-circuit des clauses.

– à la place du circuit construit pour relier chaque ensemble de sommets associé à une variable $x \in U$, nous créons 3 sommets s_x^1, s_x^2 et s_x^3 tels que $a_{s_x^1} = a_{s_x^2} = a_{s_x^3} = 0$ et un sommet p_x tel que $a_{p_x} = 2m + 1$. Nous relions par un arc de poids nul chacun des sommets s_x^1, s_x^2 et s_x^3 à p_x ainsi qu'à chacun des sommets c_x et, le cas échéant, c_x^v. Nous obtenons la structure représentée par la figure 3.15.
– comme auparavant, nous fixons pour tout arc $e \in E$ du graphe créé, $l(e) = t_{in}(e) = t_{out}(e) = 1$. Nous définissons $\lambda = 2m + 3$ et pour tout sommet $u \in V$, $\sigma(u, i) = a_u + \lambda i$. Enfin $K = 3n + 3(m - 1) + 8$.

L'ensemble des littéraux correspond à $9m$ sommets et $6m$ arcs. La structure acyclique des clauses correspond à $3 + 1$ nouveaux sommets, et chacun des sommets s^i est relié au pire à 2 sommets par littéral, donc 6 par clause plus p, donc au pire $3(6m + 1)$ arcs pour l'ensemble de la structure. De

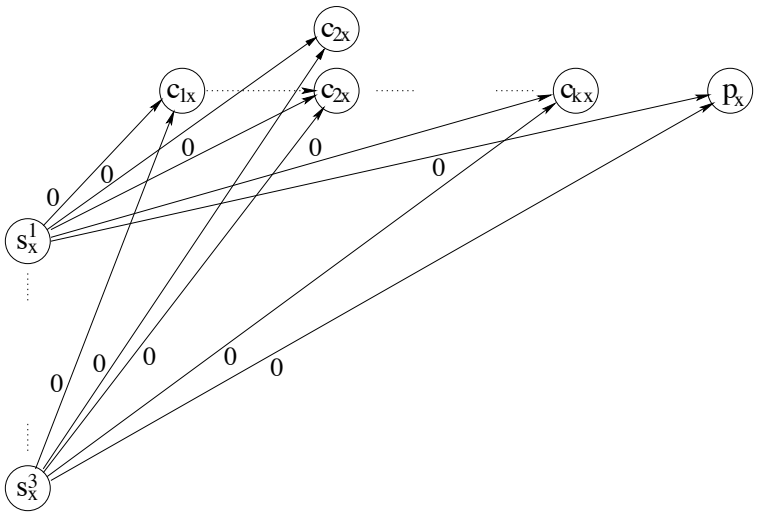


FIG. 3.15 – Structure remplaçant chaque circuit des variables.

même, chaque structure acyclique d'une variable correspond à $3+1$ nouveaux sommets et $3(6m+1)$ arcs. Donc, en tout, nous avons au pire $4n+9m+4$ sommets et $18mn+3n+24m+4$ arcs. La transformation est polynomiale en la taille de (U, C).

Remarque : La preuve suit le même principe que celle du cas cyclique, mais les circuits ont été remplacés par des structures acycliques de contrainte. Dans la preuve précédente, ces circuits imposaient à certains sommets d'avoir la même valeur de *retiming*, en jouant sur la correction de r par rapport à σ. Ici, les structures acycliques ont le même rôle mais jouent sur le coût de r : le coût sera trop grand si les sommets reliés par une même structure de contrainte n'ont pas la même valeur de *retiming*.

Remarques précédentes : Notons que les remarques sur les $s(e,t)$ et les $d_{min}(e)$ que nous avions faites dans la preuve du cas général tiennent toujours.

Réduction Soit (U, C) une instance de 3SAT et $(G, \sigma, K) = f(U, C)$. Nous devons prouver que :

- si (U, C) est une instance positive de 3SAT, alors (G, σ, K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE. Nous définissons alors notre *retiming* de façon quasi similaire au cas général. Soit T un assignement de vérité pour U satisfaisant toutes les clauses de C. Nous posons (lorsque les sommets correspondants existent dans la structure) :

$$
\begin{array}{l}
r(s^1) = r(s^2) = r(s^3) = r(p) = 0 \\
\forall c \in C, \quad \forall x \in \{x \in U \mid x \in c \text{ ou } \bar{x} \in c\}, \\
\qquad r(c_x) = r(c_x^v) = r(s_x^1) = r(s_x^2) = r(s_x^3) = r(p_x) = \left\{ \begin{array}{l} 1 \text{ si } T(x) = V \\ 0 \text{ sinon} \end{array} \right. \\
\qquad r(c_x^0) = r(c_x^c) = 0
\end{array}
$$

  Le raisonnement sur le coût d'une clause que nous avons fait pour la preuve du cas général tient toujours, mais les valeurs sont différentes. Cette fois, le coût d'une clause est 3 aux instants non locaux et 3, 4, 5 ou 6 à l'instant local selon si 3, 2, 1 ou aucun littéral est satisfait. Il ne

nous reste donc qu'à calculer le coût des nouvelles structures acycliques. Nous pouvons vérifier que, par construction de celles-ci et par définition de r, tous les arcs sortant des s^i et des s^i_x ont un poids nul, donc le coût d'un sommet s^i où s^i_x est déterminé par le coût de son arc le plus long, c'est à dire respectivement l'arc vers p ou vers p_x. Le coût de chaque structure acyclique est donc de 3 (car il y a 3 sommets sources) pour les instants $1 \leq t \leq 2m+1$ et 0 ailleurs.

Donc en faisant le total, de façon analogue au cas général, le coût final est au pire $3n+3m+3$ aux instants non locaux, et $3n+3(m-1)+3+5$ aux instants locaux. Finalement, $maxlive_r \leq K$ et (G, σ, K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE.

– si (G, σ, K) est une instance positive de RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE, alors (U, C) est une instance positive de 3SAT. Une fois que nous aurons montré que les nouvelles structures acycliques ont la même fonction que les circuits de la transformation pour le cas général, la preuve sera la même. Nous commençons par définir r un *retiming* de G, correct par rapport à σ, tel que $maxlive_r \leq K$ dans G_r.

Tout d'abord, nous devons faire une remarque sur les divers coûts de notre nouvelle transformation. Concernant les littéraux, pour chaque structure d'un littéral $l \in c$ tel que $c \in C$ et $l = x$ où $l = \bar{x}$ avec $x \in U$, comme le *retiming* est correct par rapport à σ, nous sommes sûrs que l'arc de c_x vers c^v_x ou c^0_x vers c^c_x (selon si le littéral est positif ou négatif) a un poids de 1. Nous pouvons donc supposer que, quel que soit le *retiming* (correct), l'ensemble des structures associées à une clause a un coût de 3 au minimum.

Concernant les nouvelles structures acycliques, par correction du *retiming* par rapport à σ, nous sommes sûrs que tous les arcs sortant de chacun des s^i et s^i_x ont un poids positif ou nul. Si tous ces arcs ont un poids nul, en prenant le plus long, nous obtenons un coût de $3n+3$ pour les instants $1 \leq t \leq 2m+1$. Supposons alors que pour un $s = s^i$ ou $s = s^i_x$ tous les successeurs de s n'aient pas la même valeur de *retiming*. Comme le poids de tous les arcs sortant de s est positif ou nul, cela signifie que le poids d'au moins un de ses successeurs est strictement positif, et ce raisonnement s'applique de même à toutes les autres sources (s^i ou s^i_x) de la structure acyclique – puisque ces sources ont les mêmes puits que s. Donc le coût de cette structure à l'instant 1 est supérieur ou égal à $3+3=6$ (le poids maximal plus le $s(e,t)$ à cet instant), soit au total avec le reste du graphe, un coût d'au moins $3n+3m+6 = 3n+3(m-1)+9 > K$ ce qui contredit le fait que r soit une solution. Donc, tous les successeurs d'un s^i ou s^i_x ont la même valeur de *retiming*. Finalement nous pouvons supposer que pour toute clause $c \in C$ et toute variable $x \in U$ telle que $x \in c$ ou $\bar{x} \in c$, nous avons $r(c^c_x) = r(c^0_x) = 0$. De même pour toute variable $x \in U$, si $\{c_1, \ldots, c_k\} \subset C$ est l'ensemble des clauses contenant x ou $\bar{x}$ alors $r(c_{1x}) = \cdots = r(c_{kx})$ (toutes les instances d'une variable ont la même valeur de *retiming*).

Nous définissons l'assignement de vérité comme précédemment :

$$\forall x \in U, T(x) = \begin{cases} V \text{ si } \exists c \in C, r(c_x) = 1 \\ F \text{ sinon} \end{cases}$$

Le reste de la preuve est identique au cas général, sauf que cette fois-ci nous ajoutons le coût des nouvelles structures acycliques au coût total afin de montrer qu'au moins un littéral est satisfait dans chaque clause. Finalement, (U, C) est une instance positive de 3SAT.

Ceci termine la preuve : RÉORDONNANCEMENT PAR ÉTAGES ACYCLIQUE est NP-complet au sens fort. □

3.4.3 Solution par programmation linéaire

Dans cette section, nous proposons une méthode de résolution du problème de réordonnancement par niveaux par programmation linéaire en nombres entiers. Bien entendu cette résolution est en théorie exponentielle, cependant le programme que nous donnons est relativement petit, il contient un nombre linéaire de contraintes et les expérimentations que nous avons menées à l'aide de l'outil PASTAGA (voir section 2.5 pour une présentation de PASTAGA, et section 3.5 pour les résultats des expérimentations) montrent que sa résolution est tout à fait envisageable en pratique.

Proposition 9 *Soit $G = (V, E = E_{val} \cup E_{dep})$ un graphe de dépendance pondéré sur ses arcs par d, l, t_{in} et t_{out} ainsi que $\sigma(v, k) = a_v + k\lambda$ un ordonnancement cyclique des sommets de G. Toute solution entière optimale (M, r, m) au programme suivant est telle que r est un* retiming *de G correct par rapport à σ qui minimise $maxlive_r$:*

$$
\begin{array}{l}
\text{Minimiser :} \\
\qquad M \\
\text{Tel que :} \\
\forall e = (u, v) \in E, \\
\qquad d(e) + r(v) - r(u) \geq d_{min}(e) \\
\forall t \in dates_\sigma, \\
\forall e = (u, v) \in E_{val}, \\
\qquad m(u, t) \geq d(e) + r(v) - r(u) + s(e, t) \ (= vie_r(e, t) \text{ définie en page 81}) \\
\forall t \in dates_\sigma, \\
\qquad M \geq \sum_{u \in V_{val}} m(u, t)
\end{array}
\tag{3.1}
$$

Preuve Soit (M, r, m) une solution entière optimale du programme 3.1. D'après les contraintes du programme, r vérifie $\forall e = (u, v) \in E, d_r(e) \geq d_{min}(e)$. Donc r est un *retiming* correct par rapport à σ. De plus, à l'optimal, $M = \max_{t \in dates_\sigma} \sum_{u \in V_{val}} m(u, t)$ et que $\forall t \in dates_\sigma, m(u, t) = \max_{e=(u,v) \in E_{val}} vie_r(e) = valeurs_r(u, t)$, sinon il suffit de prendre ces valeurs pour obtenir une meilleure solution. Donc $M = maxlive_r$, et comme le programme minimise M, r est un *retiming* minimisant $maxlive_r$. □

Nous pouvons vérifier que le nombre de variables de ce programme est $|V_{val}||dates_\sigma| + |V| + 1$ et son nombre de contraintes $|E_{val}||dates_\sigma| + |dates_\sigma| + |E|$, c'est-à-dire deux quantités polynomiales en la taille de notre graphe. À l'inverse, Eichenberger et al. proposent dans [36] de résoudre le problème par un nombre exponentiel de programmes linéaires contenant un nombre exponentiel de contraintes.

3.4.4 Comparaison avec la méthode d'Eichenberger et Al.

Nous détaillons dans cette section les améliorations que notre solution apporte par rapport à celle d'Eichenberger, Davidson et Abraham présentée dans [36]. Notre solution apporte des améliorations aussi bien dans l'expression des contraintes de dépendances que dans l'expression du coût du décalage.

Dans leur méthode, Eichenberger, Davidson et Abraham ne présentent pas le décalage des instructions comme un *retiming*, mais comme une distance additionnelle à ajouter au poids des arcs – distance qu'ils appellent *skip factor*. Dans ce contexte, le décalage n'est donc plus une pondération des sommets mais une pondération des arcs. Le problème est que, pour assurer la cohérence du décalage, il faut être sûr, entre autres, que si deux chemins distincts existent entre deux sommets dépendants, alors ils ont le même poids. Dans le cas contraire, cela signifierait que la distance

entre les deux instances de l'opération source dont dépend l'opération puits a changé, ce qui est impossible puisque le motif de l'ordonnancement cyclique est resté le même. De façon plus générale, bien qu'Eichenberger, Davidson et Abraham ne le démontrent pas formellement, cela correspond à la propriété de conservation du poids d'un cycle par *retiming* (voir la proposition 1). Ainsi, il se voient forcés d'imposer la cohérence de leur distance additionnelle en ajoutant une contrainte au programme linéaire pour chaque cycle du graphe (qu'ils appellent *underlying cycle*), soit en tout un nombre potentiellement exponentiel en fonction de la taille du graphe. Dans notre formulation, la cohérence est assurée simplement par la propriété naturelle du *retiming* de conservation du poids de cycles, nous n'avons donc aucun besoin de ces contraintes. Donc, Eichenberger, Davidson et Abraham choisissent de représenter le décalage des instructions sur les arcs, en tant que variation de la distance de dépendance des arcs, ou *skip factor*. Malheureusement, ce choix de modèle est une erreur qui les conduit à résoudre un nombre potentiellement exponentiel de contraintes dans chaque programme linéaire qu'ils auront à considérer.

Afin de proposer une méthode de résolution optimale, Eichenberger, Davidson et Abraham font une remarque sur l'ordre de consommation des valeurs : pour toute valeur produite, il existe une instruction consommant cette valeur en dernier et à partir de laquelle le registre peut être libéré. Cette propriété n'est pas démontrée formellement dans [36], cependant il suffit d'examiner $cons_r(e,t)$ pour la prouver. C'est ce que nous nous proposons de faire dans la proposition suivante.

Proposition 10 *Soit un graphe de dépendance G, un* retiming *r de G et un ordonnancement cyclique $\sigma(v,i) = a_v + \lambda(i + r(v))$ des opérations de G. Pour toute instruction $u \in V_{val}$, il existe un successeur de u, $v \in \{v \in V \mid e = (u,v) \in E_{val}\}$ tel que*

$$\forall v' \in \{v' \in V \mid e' = (u,v') \in E_{val}\}, \forall t \in [0,\ldots,\lambda[, vie_r(e,t) \geq vie_r(e',t)$$

Preuve Commençons par examiner la fonction $cons_r(e,t)$, nous avons

$$cons_r(e,t) = \left\lfloor \frac{t - a_v - t_{in}(e)}{\lambda} \right\rfloor - d(e) - r(v)$$

Soit $a_v + t_{in}(e) = \delta_e \lambda + \gamma_e$ avec $0 \leq \gamma_e < \lambda$ (division euclidienne), et $0 \leq t < \lambda$, nous obtenons :

$$cons_r(e,t) = \delta_e - d(e) - r(v) - \begin{cases} 0 \text{ si } \gamma_e \leq t \\ 1 \text{ sinon} \end{cases}$$

Soient trois sommets u,v et v' tels que $e = (u,v)$ et $e' = (u,v')$ appartiennent à E_{val}, nous avons pour tout $0 \leq t < \lambda$:

- $cons_r(e,t) \geq cons_r(e',t)$ si $\delta_e - d(e) - r(v) > \delta_{e'} - d(e') - r(v')$ ou si $\delta_e - d(e) - r(v) = \delta_{e'} - d(e') - r(v')$ et $\gamma_e \leq \gamma_{e'}$;
- $cons_r(e,t) \leq cons_r(e',t)$ sinon.

Autrement dit, la condition pour que $cons_r(e,t) \geq cons_r(e',t)$ est indépendante de t : le dernier consommateur d'une valeur est toujours le même. Donc, pour tout sommet $u \in V_{val}$, il existe $v \in \{v \in V \mid e = (u,v) \in E_{val}\}$ tel que pour tout $v' \in \{v' \in V \mid e' = (u,v') \in E_{val}\}$:

$$\forall t \in [0,\ldots,\lambda[, cons_r(e,t) \leq cons_r(e',t)$$

c'est-à-dire :

$$\forall t \in [0,\ldots,\lambda[, vie_r(e,t) \geq vie_r(e',t)$$

□

L'idée employée par Eichenberger, Davidson et Abraham est alors la suivante : si nous connaissons, pour chaque sommet $u \in V_{val}$, le bon successeur $v_u \in \{v \in V \mid e_u = (u, v_u) \in E_{val}\}$, alors il suffit d'imposer $vie_r(e_u, t) \geq vie_r(e, t)$ pour tout autre successeur $v \in \{v \in V \mid e = (u, v) \in E_{val}$ et $v \neq v_u\}$ et de minimiser $\sum_{u \in V_{val}} vie_r(e_u)$ pour trouver une solution optimale au problème de réordonnancement par niveaux. Notons que dans ce cas, le programme linéaire associé possède une matrice totalement unimodulaire et peut donc être résolu en temps polynomial[3]. Mais évidemment, au préalable, il faut connaître le bon successeur pour chaque sommet...

Dans leur article, Eichenberger, Davidson et Abraham prétendent que le nombre de combinaisons de sommets successeurs à essayer est très réduit, et qu'il suffit d'essayer uniquement les permutations les plus « attractives » pour atteindre l'optimal[4]. Notre résultat de NP-complétude prouve que, à moins que P=NP, le nombre de cas à essayer est effectivement exponentiel, et qu'il n'y a aucun moyen de savoir que l'optimal est atteint, excepté en essayant toutes les permutations possibles de sommets successeurs.

Autrement dit, utiliser cette méthode de résolution revient à payer systématiquement la complexité du pire cas, puisqu'un nombre exponentiel de programmes doit être résolu. Il semblerait qu'Eichenberger et al. n'aient pas réalisé qu'un nombre réduit de dates est à considérer pour le calcul de *maxlive*, ni que *maxlive* peut s'exprimer directement sous forme de contraintes simples dans le programme linéaire. Notre programme comportant au plus $|V_{val}||E_{val}|$ contraintes pour exprimer le coût permet de trouver le *retiming* optimal en une seule résolution. Ceci lui permet de profiter pleinement de l'excellente efficacité pratique de la programmation linéaire (même si la résolution en nombres entiers est, en théorie, exponentielle).

3.4.5 Une approximation polynomiale

La complexité du réordonnancement par étage a un côté déconcertant ; en effet, des problèmes de *retiming* proches comme le partage maximal de registres (voir [70]) sont résolus efficacement par des variantes d'algorithmes de flot de coût minimal. Dans cette section, nous montrons qu'en faisant une légère approximation sur le nombre de valeurs générées par un arc, nous pouvons faire disparaître l'aspect combinatoire du problème. Il en résulte un algorithme polynomial pour minimiser cette approximation du coût, algorithme qui, comme celui du chapitre 2, est une variante d'un problème de recherche d'un flot de coût minimal dans le graphe.

Revenons tout d'abord à notre fonction $vie_r(e, t)$. En reprenant le raisonnement de la proposition 6 de la page 82, nous obtenons :

$$vie_r(e, t) = d_r(e) - \alpha_e + \delta_e + s'(e, t)$$

avec $a_u + t_{out}(e) = \alpha_e \lambda + \beta_e$ où $0 \leq \beta_e < \lambda$ et $a_v + t_{in}(e) = \delta_e \lambda + \gamma_e$ où $0 \leq \gamma_e < \lambda$ (division euclidienne). Dans ce cas, $s'(e, t) = -1$, 0 ou 1, selon les valeurs de t, β_e et γ_e (voir proposition 6). Plus intéressant, quelle que soit la valeur de t, nous avons les cas suivants :

- si $\beta_e = \gamma_e$ alors $s'(e, t) = 0$ et nous posons $s_{min}(e) = 0$;
- si $\beta_e < \gamma_e$ alors $s'(e, t) = 0$ ou 1 et nous posons $s_{min}(e) = 0$;
- si $\beta_e > \gamma_e$ alors $s'(e, t) = -1$ ou 0 et nous posons $s_{min}(e) = -1$.

[3] Sauf que, pour Eichenberger et al., le nombre de contraintes est potentiellement exponentiel. Le temps de résolution est donc polynomial en une quantité exponentielle... En revanche, l'usage du *retiming* permet d'avoir une résolution réellement polynomiale.

[4] *Interesting sets to be evaluated consist of all possible forcing combinations of one attractive last-use for some or all virtual registers whose fractional lifetime can be decreased. [...] we have to consider only those cases that potentially reduce the fractional Maxlive. We do not need to consider use operations unless they may impact the MRT row that could govern the second term of Eq. (16).*

Nous obtenons :

$$\forall 0 \leq t < \lambda, s_{min}(e) \leq s'(e,t) \leq s_{min}(e) + 1 \tag{3.2}$$

et donc :

$$survie_r(e) = d_r(e) - \alpha_e + \delta_e + s_{min}(e)$$

est une approximation de $vie_r(e,t)$ à -1 près ne dépendant pas de t. Si nous exprimons le coût en fonction de cette quantité, nous pouvons alors remplacer notre programme linéaire 3.1 par le programme suivant :

$$\begin{array}{l} \text{Minimiser :} \\ \qquad M = \sum_{u \in V_{val}} m(u) \\ \text{Tel que :} \\ \forall e = (u,v) \in E, \\ \qquad d(e) + r(v) - r(u) \geq d_{min}(e) \\ \forall e = (u,v) \in E_{val}, \\ \qquad m(u) \geq survie_r(e) \end{array} \tag{3.3}$$

À partir de l'équation 3.2, nous déduisons que le coût d'une solution optimale de 3.3 est inférieur ou égal au coût d'une solution optimale du programme 3.1 de la page 97. De plus, toujours à partir de l'équation 3.2, nous déduisons que toute solution (M, r, m) du programme 3.3 correspond à un *retiming* dont le coût pour le programme 3.1 est au pire de $M + |V_{val}|$. Donc la résolution du programme 3.1 nous donne une solution approchée à $|V_{val}|$ près au problème du réordonnancement par étages.

Par ailleurs, nous pouvons vérifier que cette borne est atteinte. Pour s'en convaincre, il suffit d'examiner la figure 3.12 de la page 94, en ignorant les arcs de contrainte des clauses et des variables. Cette figure est composée de trois sommets : c_x, c_x^v et c_x^0. Parmi ces trois sommets, c_x^v et c_x^0 ne font pas partie de V_{val}, car ils n'ont pas d'arc sortant du tout. Le coût du *retiming* est donc entièrement déterminé par le sommet c_x. À cause de l'arc de c_x vers c_x^v et de son d_{min}, le coût de c_x est toujours au minimum de 1. L'arc de c_x vers c_x^0 a un $s(e,t)$ égal à 0 ou 1 selon l'instant considéré, donc il est approximé par $s_{min}(e) = 0$. Donc le coût d'un *retiming* nul pour notre approximation est de 1. C'est donc une solution optimale au programme 3.3. Pourtant, le coût réel du *retiming* nul est de 2, notre approximation ne perçoit pas la nécessité d'effectuer un *retiming* à cause de la valeur de $s(e,t)$. En ajoutant suffisamment de copies de cette structure ordonnancées en parallèle, nous pouvons montrer que la borne est atteinte pour toute valeur de $|V_{val}|$.

Notons que nous pouvons aussi faire une approximation par excès de $s'(e,t)$, en définissant :

$$survie_r(e) = d_r(e) - \alpha_e + \delta_e + s_{max}(e)$$

où $s_{max}(e) = 1$ si $\beta_e < \gamma_e$, et $s_{max}(e) = 0$ sinon. Dans ce cas, de façon analogue, nous prouvons que nous obtenons une approximation à $|V_{val}|$ près du problème de réordonnancement par étages. Nous avons pu constater expérimentalement une différence significative de performance entre les deux approches. En effet, l'approximation par excès de $s'(e,t)$ donne de façon quasi-systématique des résultats meilleurs que l'approximation par défaut. Cependant, d'un point de vue théorique, nous ne sommes pas en mesure de donner une analyse aussi précise de l'approximation par excès, car nous ignorons si elle atteint la borne.

Revenons à la résolution de notre programme 3.3. Nous pouvons remarquer que ce problème est très proche du problème de maximisation du partage des registres résolu par Leiserson et Saxe dans [70]. Les notations sont différentes, mais à l'arrivée ils minimisent $\sum_{u \in V} \max_{e \in E} d_r(e)$. Ils

résolvent leur problème par un algorithme de flot en nombres rationnels mais leur solution semble difficile à modifier afin de changer la fonction de coût et les contraintes de légalité. Plus intéressant, le programme linéaire que nous obtenons est presque identique à celui proposé par Ning et Gao dans [82]. Les seules différences sont que, pour eux, l'ordonnancement est également une variable du problème et leur coût est différent : il ne permet pas de donner la même borne de performance pour la solution. En revanche, nous pouvons utiliser la même technique que Ning et Gao pour résoudre notre problème : tout d'abord prouver qu'il est polynomial, et ensuite le remplacer par un algorithme de flot en nombre entiers[5].

Montrons tout d'abord que le programme 3.3 peut être résolu en temps polynomial, et donc que nous pouvons calculer notre approximation plus efficacement que la solution exacte. Pour cela, il nous faut l'exprimer sous forme matricielle :

$$\min \left\{ \begin{pmatrix} r & m \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} \middle| \begin{pmatrix} r & m \end{pmatrix} \begin{pmatrix} -C & C_{val} \\ 0 & C^+_{val} \end{pmatrix} \geq \begin{pmatrix} d_{dep} & d_{val} \end{pmatrix} \right\} \tag{3.4}$$

où C est la matrice de connexion de notre graphe G (une matrice $|V|\times|E|$ où chaque colonne contient exactement un 1, un -1, correspondant respectivement aux sommets source et destination de chaque arc, et toutes les autres entrées à 0, voir [38]), C_{val} la matrice de connexion de $G_{val} = (V, E_{val})$, C^+_{val} la matrice déduite de C_{val} en remplaçant toutes les entrées négatives par des 0 et en enlevant toutes les lignes ne contenant aucune entrée positive, et où $\forall e \in E, d_{dep}(e) = d_{min}(e) - d(e)$ et $\forall e \in E_{val}, d_{val}(e) = d(e) - \alpha_e + \delta_e + s_{min}(e)$ (ou $s_{max}(e)$, si nous souhaitons faire une approximation par excès).

Proposition 11 *Le programme linéaire 3.4 a une solution optimale entière qui peut être trouvée en temps polynomial.*

Preuve Nous allons prouver que la matrice du programme 3.4 est totalement unimodulaire, le reste est assuré par la théorie de la programmation linéaire (voir [99]). La matrice du programme :

$$A = \begin{pmatrix} -C & C_{val} \\ 0 & C^+_{val} \end{pmatrix}$$

est totalement unimodulaire si et seulement si toutes ses sous-matrices carrées sont unimodulaires (de déterminant 1, -1 ou 0). Considérons une sous-matrice carrée A' de A. Si elle contient des lignes de la sous matrice $\begin{pmatrix} 0 & C^+_{val} \end{pmatrix}$, alors pour chacune de ces lignes l :

- soit la ligne correspondante l' de $\begin{pmatrix} -C & C_{val} \end{pmatrix}$ est présente dans A' (par définition, chacune des lignes de C^+_{val} correspond à une ligne de C_{val} comportant les mêmes entrées positives). Dans ce cas, nous pouvons soustraire l à l', ce qui est une transformation unimodulaire de A' et ne change donc pas son déterminant. Comme C_{val} contient au plus un seul 1 par colonne, après la soustraction, toutes les entrées positives de l sont les seules dans leur colonne.
- soit la ligne correspondante de $\begin{pmatrix} -C & C_{val} \end{pmatrix}$ n'est pas présente dans A' et dans ce cas toutes les entrées positives de l sont les seules dans leur colonne (car ce sont aussi les entrées positives d'une ligne de C_{val} qui n'est pas dans A' et, par définition d'une matrice de connexion, il n'y en a pas d'autres).

Dans tous les cas, nous aboutissons finalement à une matrice A' et à une éventuelle transformation unimodulaire telle que chaque colonne contient au plus un seul 1 et un seul -1 (et toutes les autres

[5] Les algorithmes de flot ont une meilleure complexité en nombres entiers qu'en rationnels. En effet ceci joue sur le nombre d'étapes vers la convergence.

entrées à 0). Cette dernière matrice est unimodulaire car c'est une sous-matrice d'une matrice de connexion (et les matrices de connexion sont totalement unimodulaires, voir [99]). □

Nous pourrions nous contenter de ce résultat et recourir à un solveur de programmes linéaires pour résoudre notre problème d'approximation, mais nous pouvons continuer notre analyse et donner une formulation du problème sous forme de recherche de flot entier de coût minimal (dont la complexité est généralement moindre que celle des algorithmes de résolution de programmes linéaires). Pour ceci, nous avons besoin d'exprimer le dual de notre programme (voir [28, p. 33]) :

$$\max \left\{ \begin{pmatrix} d_{dep} & d_{val} \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} \middle| \begin{pmatrix} -C & C_{val} \\ 0 & C_{val}^{+} \end{pmatrix} \begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}, \begin{pmatrix} x & y \end{pmatrix} \geq 0 \right\} \tag{3.5}$$

Le théorème de la dualité de la programmation linéaire nous dit que le programme primal et le programme dual ont des solutions optimales de même coût, et le fait que la matrice du programme soit totalement unimodulaire garantit que les deux programmes ont une solution optimale entière. Réécrit sous une forme plus classique, le programme 3.5 devient :

Minimiser : (3.6)

$$-\sum_{e \in E} d_{dep}(e)x(e) - \sum_{e \in E_{val}} d_{val}(e)y(e)$$

Tel que :

$\forall u \in V$,

$$\sum_{e=(.,u) \in E} x(e) - \sum_{e=(.,u) \in E_{val}} y(e) = \sum_{e=(u,.) \in E} x(e) - \sum_{e=(u,.) \in E_{val}} y(e) \tag{3.7}$$

$\forall u \in V_{val}$,

$$\sum_{e=(u,.) \in E_{val}} y(e) = 1 \tag{3.8}$$

$$\begin{array}{ll} \forall e \in E, & x(e) \geq 0 \\ \forall e \in E_{val}, & y(e) \geq 0 \end{array} \tag{3.9}$$

La première contrainte 3.7 rappelle fortement une contrainte de flot. Elle laisse penser que nous souhaitons trouver deux valeurs $x(e)$ sur tous les arcs et $y(e)$ sur les arcs de E_{val} (que nous pouvons étendre en posant $y(e) = 0$ sur les arcs $e \notin E_{val}$) telles que $x(e) - y(e)$ soit un flot de G. La deuxième contrainte 3.8 est une contrainte structurelle sur la composante $y(e)$ du flot : combinée aux contraintes de positivité 3.9 et au fait que nous pouvons nous restreindre aux solutions entières, elle impose à tout sommet $u \in V_{val}$ d'avoir une valeur $y(e)$ égale à 1 pour exactement un seul arc e et à 0 pour tous les autres arcs. Enfin, la dernière contrainte 3.9 va nous conduire à chercher des flots positifs dans notre graphe. Il nous reste donc à transformer notre graphe $G = (V, E, d)$ en $G_f = (V_f, E_f, d_f)$ (voir figure 3.16) pour que toutes ces contraintes s'expriment naturellement comme des contraintes classiques de flot :

- nous partons de $G_f = G$, pour tous les arcs $e \in G_f, d_f(e) = -d_{dep}(e)$ et nous fixons une capacité inférieure de flot à $l(e) = 0$ (positivité du flot) et une capacité supérieure à $u(e) = +\infty$ (pas de capacité supérieure) ;
- pour tout sommet $u \in V_{val}$, nous ajoutons à G_f un nouveau sommet v_u et un arc e_u de v_u vers u de poids nul, de capacité inférieure de flot $l(e_u) = 1$ et de capacité supérieure $u(e_u) = 1$;
- pour tout arc $e = (u, v) \in E_{val}$, nous ajoutons à G_f un nouvel arc e_f de v vers v_u de poids $d_f(e) = -d_{val}(e)$, de capacité inférieure de flot $l(e) = 0$ et de capacité supérieure $u(e) = +\infty$.

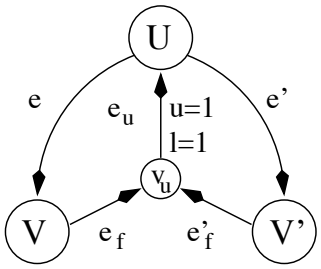


FIG. 3.16 – Transformation du graphe.

Proposition 12 *Un flot positif f de G_f a un coût de M si et seulement si (x, y) est une solution admissible du programme 3.6 de coût M avec $\forall e \in E, x(e) = f(e)$ et $\forall e \in E_{val}, y(e) = f(e_f)$.*

Preuve Pour chaque sommet $u \in V_f$, comme f est un flot, nous avons :

$$\sum_{e=(u,.)\in E_f} f(e) = \sum_{e=(.,u)\in E_f} f(e) \tag{3.10}$$

Par construction de G_f, pour tout sommet $u \in V_{val}$, e_u est le seul arc entrant dans u que nous ayons ajouté à G. De plus, pour tout arc $e = (u, v) \in E_{val}$, nous avons un arc $e_f = (v, v_u)$ sortant de v (autrement dit de direction opposée), donc pour tous les sommets $u \in V$ nous pouvons réécrire l'équation 3.10 de la façon suivante :

$$\sum_{e=(u,.)\in E} f(e) + \sum_{e=(.,u)\in E_{val}} f(e_f) = \sum_{e=(.,u)\in E} f(e) + f(e_u) \tag{3.11}$$

À cause des capacités, pour tout flot admissible f de G_f, nous avons $f(e_u) = 1$, donc en utilisant la conservation du flot aux nœuds de type v_u, nous obtenons pour tous les nouveaux sommets :

$$\sum_{e=(u,.)\in E_{val}} f(e_f) = f(e_u) = 1 \tag{3.12}$$

Donc, en définissant $\forall e \in E, x(e) = f(e)$ et $\forall e \in E_{val}, y(e) = f(e_f)$, nous obtenons :

– en utilisant l'équation 3.12 pour remplacer $f(e_u)$ par $\sum_{e=(u,.)\in E_{val}} f(e_f)$ dans l'équation 3.11 :

$$\forall u \in V,$$
$$\sum_{e=(u,.)\in E} x(e) - \sum_{e=(u,.)\in E_{val}} y(e) = \sum_{e=(.,u)\in E} x(e) - \sum_{e=(.,u)\in E_{val}} y(e)$$

– en utilisant l'équation 3.12 :

$$\forall u \in V_{val},$$
$$\sum_{e=(u,.)\in E_{val}} y(e) = 1$$

– en utilisant la positivité du flot f :

$$\begin{array}{l} \forall e \in E, \\ \qquad x(e) \geq 0 \\ \forall e \in E_{val}, \\ \qquad y(e) \geq 0 \end{array}$$

Autrement dit, (x, y) est une solution admissible du programme 3.6. Nous pouvons vérifier que toutes les étapes de la démonstration sont des équivalences et donc, que la réciproque est vraie. En outre, nous avons :

$$\sum_{e \in E_f} d_f(e) f(e) = -\sum_{e \in E} d_{dep}(e) x(e) - \sum_{e \in E_{val}} d_{val}(e) y(e)$$

Donc f et (x, y) ont le même coût. □

Une solution optimale du programme 3.6 peut donc être déterminée en trouvant un flot de coût minimal dans G_f. Nous ne détaillerons pas ici d'algorithme de recherche d'un flot de coût minimal dans un graphe, nous invitons le lecteur à se reporter aux nombreux ouvrages sur le sujet. La plupart des algorithmes de recherche de flot réclament un flot initial admissible. Ceci se fait de manière immédiate pour G_f en choisissant pour chaque sommet $u \in V_{val}$ un arc sortant $e = (u, v) \in E_{val}$ et en faisant passer une unité de flot à travers le circuit $\{(u, v), (v, v_u), (v_u, u)\}$. Toutes les capacités sont alors satisfaites. La résolution pourra être faite par un algorithme du type de l'algorithme des arcs non conformes (*out of kilter*, dont une variante a déjà été utilisée au chapitre 2, voir [38]). Dans ce cas, la solution optimale du problème primal est déterminée en même temps, et nous obtenons donc notre *retiming*. Dans le cas contraire, si nous ne disposons que de la solution du dual, nous devons disposer d'un moyen pour retrouver la solution du primal. Pour cela, nous devons rappeler la définition du graphe d'écart (voir [28]).

Définition 22 (graphe d'écart) *Soit $G = (V, E, d)$ un graphe de dépendance muni de capacités inférieure l et supérieure u sur ses arcs et un flot admissible f de G, nous définissons $R_G(f) = (V, E_r, d_r)$ le graphe d'écart de G de la façon suivante :*

- *nous partons de $R_G(f) = (V, \emptyset)$;*
- *pour tout arc $e = (u, v) \in E$:*
  - *si $f(e) < u(e)$ nous ajoutons à $R_G(f)$ un arc de u vers v de poids $d(e)$;*
  - *si $f(e) > l(e)$ nous ajoutons à $R_G(f)$ un arc de v vers u de poids $-d(e)$.*

(Selon la valeur de $f(e)$ par rapport à $l(e)$ et $u(e)$ il y a donc aucun, un ou deux arcs entre u et v dans $R_G(f)$).

Proposition 13 *Soit un graphe de dépendance G et un flot de coût minimal de G_f, alors pour tout* retiming *légal du graphe d'écart $R_{G_f}(f)$, il existe m et M tels que (M, r, m) soit une solution optimale du programme 3.3.*

Preuve Puisque f est de coût minimal dans G_f, le graphe d'écart $R_{G_f}(f)$ n'a pas de circuit de poids strictement négatif (voir [28], mais la preuve est immédiate : si un tel circuit existe, il suffit de modifier le flot sur les arcs correspondants, +1 sur les arcs de G_f dans le même sens et -1 sur les autres ; par construction de $R_{G_f}(f)$ le flot reste alors admissible mais son coût diminue), donc à l'aide de l'algorithme 2 nous pouvons trouver en temps polynomial un *retiming* légal de $R_{G_f}(f)$.

Par construction de G_f, tous les arcs $e = (u, v) \in E$ de G sont dans G_f, et comme ils n'ont pas de capacité supérieure, ils sont également dans le graphe d'écart de G_f (et ont le même poids que

dans G_f). Autrement dit, pour tout arc $e = (u, v) \in E$, $R_{G_f}(f)$ contient un arc de u vers v de poids $-d_{dep}(e)$, et comme r est légal dans $R_{G_f}(f)$, nous avons :

$$\begin{aligned} & -d_{dep}(e) + r(v) - r(u) \geq 0 \\ \iff \quad & d(e) + r(v) - r(u) \geq d_{min}(e) \end{aligned} \tag{3.13}$$

Nous définissons alors m de la manière suivante :

$$\forall u \in V_{val}, m(u) = \max_{e=(u,v)\in E_{val}} d_{val}(e) + r(v) - r(u)$$

et

$$M = \sum_{u \in V_{val}} m(u)$$

Ainsi, par construction de m et M et par l'équation 3.13, nous déduisons que (M, r, m) est une solution admissible du programme 3.1.

Il reste à prouver que (M, r, m) est optimale. Pour cela, nous considérons la solution optimale (x, y) du programme 3.6 construite à partir de f comme indiqué dans la proposition 12, et nous appliquons le théorème des écarts complémentaires (voir [28, page 39]). Nous devons alors prouver que les inégalités suivantes sont vérifiées :

$$(-rC + -d_{dep})x = 0 \tag{3.14}$$

$$(rC_{val} + mC^{+}_{val} - d_{val})y = 0 \tag{3.15}$$

Soit un arc $e = (u, v) \in E$, nous avons les deux possibilités suivantes :

- si $f(e) = 0$ alors $x(e) = 0$ et $(r(v) - r(u) - d_{dep}(e))x(e) = 0$;
- sinon, par construction de G_f, puisque e n'a pas de capacité supérieure de flot, $R_{G_f}(f)$ contient un arc de u vers v de poids $-d_{dep}(e)$ et un arc de v vers u de poids $d_{dep}(e)$. Comme r est un *retiming* légal de $R_{G_f}(f)$, nous avons :

$$\begin{aligned} -d_{dep}(e) + r(v) - r(u) &\geq 0 \\ d_{dep}(e) + r(u) - r(v) &\geq 0 \end{aligned}$$

  donc $r(v) - r(u) = d_{dep}(e)$ et $(r(v) - r(u) - d_{dep}(e))x(e) = 0$.

Donc $\forall e(u, v) \in E, (r(v) - r(u) - d_{dep}(e))x(e) = 0$, et la contrainte 3.14 est vérifiée.
Soit un arc $e = (u, v) \in E_{val}$, nous avons les deux possibilités suivantes :

- si $f(e_f) = 0$ alors $y(e) = 0$ et $(m(u) - (d_{val}(e) + r(v) - r(u)))y(e) = 0$;
- sinon, par construction des arcs e_f de G_f (un arc $e_f = (., v_u)$ correspond à un arc $e = (u, .) \in E_{val}$) :

$$\sum_{e=(u,.)\in E_{val}} f(e_f) = 1$$

  et le flot f est positif, ce qui signifie que $f(e_f) = 1$. Donc pour tout $e' = (u, v') \in E_{val}$ tel que $e \neq e'$ nous avons $f(e'_f) = 0$. Donc par construction, $R_{G_f}(f)$ contient (entre autres) un arc de v_u vers v de poids $d_{val}(e)$ et un arc de v' vers v_u de poids $-d_{val}(e')$. Comme r est légal, ceci nous donne :

$$\begin{aligned} d_{val}(e) + r(v) - r(v_u) &\geq 0 \\ -d_{val}(e') + r(v_u) - r(v') &\geq 0 \end{aligned}$$

donc :

$$\begin{aligned} d_{val}(e) + r(v) &\geq d_{val}(e') + r(v') \\ d_{val}(e) + r(v) - r(u) &\geq d_{val}(e') + r(v') - r(u) \end{aligned}$$

ce qui nous permet de déduire que $m(u) = d_{val}(e) + r(v) - r(u)$, donc $(m(u) - (d_{val}(e) + r(v) - r(u)))y(e) = 0$.

donc $\forall e = (u, v) \in E_{val}, (m(u) - (d_{val}(e) + r(v) - r(u)))y(e) = 0$, et la contrainte 3.15 est vérifiée. □

Finalement, il nous suffit donc de trouver un flot de coût minimal f dans G_f et un *retiming* légal r de son graphe d'écart $R_{G_f}(f)$, et nous obtenons un *retiming* r qui minimise :

$$\sum_{u \in V_{val}} \max_{e \in E_{val}} survie_r(e)$$

c'est-à-dire une solution approchée à $|V_{val}|$ au problème du réordonnancement par étages.

3.5 Expérimentations

Nous avons repris les configurations de machines et de graphes de la section 2.5 afin d'évaluer le gain en registres et le temps d'exécution de nos techniques de réordonnancement par étages. Chacune de nos mesures est toujours faite par une moyenne du résultat obtenu sur un échantillon de 1000 graphes aléatoires. Les graphes sont générés avec les mêmes paramètres qu'indiqué au début de la section 2.5 ($1,8$ pour le degré entrant, 10 sommets, 67% de poids nuls et 33% de poids 1). Une fois le pipeline logiciel réalisé, nous avons mesuré le *maxlive* obtenu dans les quatre situations suivantes :

- aucun traitement particulier n'est fait ;
- une approche naïve, variante de l'algorithme 2 de la page 22, tente de mettre le graphe dans un état le plus proche possible de son état initial (avant pipeline logiciel) ;
- notre approximation polynomiale par excès est utilisée ;
- notre approche exacte par programmation linéaire est utilisée. Pour cela, nous avons recours au solveur PIP [44].

Tout d'abord nous reprenons la configuration qui nous a servi à établir la table 2.1, c'est à dire une simple unité fonctionnelle pipelinée. Nous obtenons pour *maxlive* les résultats présentés en table 3.1. Comme nous pouvons le constater, pour Comp, Clock et GS, peu de gains sont à espérer.

	Comp	Clock	MinEdge	ClockEdge	GS
Aucune modification	6.08	7.08	9.74	9.78	6.89
Approche naïve	6.08	6.89	7.77	7.73	6.7
Approximation polynomiale	6.07	6.87	7.72	7.68	6.67
Approche exacte	6.04	6.83	7.68	7.65	6.64

TAB. 3.1 – *Maxlive* pour une unité fonctionnelle pipelinée.

En revanche pour MinEdge et ClockEdge les gains sont plus importants, mais les trois approches de réduction de *maxlive* s'en tirent à peu près aussi bien.

Nous avons alors essayé une autre configuration : la machine VLIW avec laquelle nous avons réalisé la table 2.7. Nous obtenons pour *maxlive* les résultats présentés en table 3.2. Les « NA »

présents dans la table indiquent que, en raison de notre heuristique d'augmentation des poids dans les parties acycliques du graphe, le résultat du pipeline logiciel n'est pas exploitable sans appliquer au minimum l'approche naïve de correction (le *maxlive* est démesuré). Encore une fois, les trois approches de réduction donnent des résultats comparables. Le plus étonnant est que l'approche naïve s'en sorte aussi bien.

	Comp	Clock	MinEdge	ClockEdge	GS
Aucune modification	6.02	6.04	NA	NA	6.14
Approche naïve	6.01	6.04	8.5	8.58	6.13
Approximation polynomiale	6	6.03	8.45	8.53	6.12
Approche exacte	5.98	6	8.43	8.51	6.09

TAB. 3.2 – *Maxlive* pour notre machine VLIW.

Pour comprendre ce phénomène, il faut réexaminer l'idée de l'approche naïve : elle tente de mettre le graphe dans un état le plus proche possible de son état initial. Si cette approche fonctionne dans nos exemples c'est simplement parce que le graphe de départ – avec beaucoup de poids nuls et le reste à 1 – n'autorise que peu d'améliorations, et produit un *maxlive* déjà presque optimal lors de l'utilisation de techniques de pipeline logiciel simples (Comp par exemple).

Pour comprendre en quoi l'approximation polynomiale est meilleure, il suffit de générer une situation dans laquelle beaucoup d'améliorations sont possibles sur le graphe de départ : nous considérons donc la même architecture mais avec un graphe comportant des poids de 5, 3, 1 et 0, chacun dans une proportion de 25%, ainsi que 20 sommets. Nous obtenons pour *maxlive* les résultats présentés en table 3.3. Nous pouvons alors voir clairement la différence entre l'approche naïve et

	Comp	Clock	MinEdge	ClockEdge	GS
Aucune modification	52.9	52.6	52.8	52.7	51.4
Approche naïve	52.9	52.6	52.7	52.7	50.2
Approximation polynomiale	44.4	44.5	46.4	46.4	43.6
Approche exacte	43.2	43.3	45.6	45.6	42.6

TAB. 3.3 – *Maxlive* pour notre machine VLIW (graphe initial améliorable).

l'approximation polynomiale : la première n'est qu'une heuristique de correction des effets négatifs du pipeline logiciel. C'est la raison pour laquelle elle ne peut pas améliorer une situation initiale déjà mauvaise. La seconde, en revanche, est une véritable heuristique de minimisation de *maxlive*. Dans tous les cas elle produira un bon résultat (à $|V_{val}|$ de l'optimal au pire).

Enfin, afin de juger de la pertinence de l'approche exacte par programmation linéaire par rapport à l'approximation polynomiale, nous devons avoir une idée de son coût. Pour cela nous avons relancé les tests de la table 3.2 pour différentes tailles de graphe. Nous obtenons, dans le cas de l'heuristique Comp, les temps moyens de calcul représentés sur la figure 3.17. Comme nous pouvons le constater, l'approche exacte a une complexité empirique dont la croissance est effectivement exponentielle. Au-delà de 20 sommets, PIP s'arrête à cause de dépassement de capacité pour les variables entières du programme. En comparaison, la complexité empirique de l'approximation polynomiale est extrêmement bonne (légèrement plus élevée qu'une croissance linéaire). Vu la qualité des résultats de cette dernière approche, son usage semble préférable à celui de l'approche exacte, qui serait plutôt à réserver aux applications critiques pour lesquelles le temps de compilation importe peu. Notons que dans d'autres situations (autres heuristiques, autres modèles de machine, autres paramètres pour

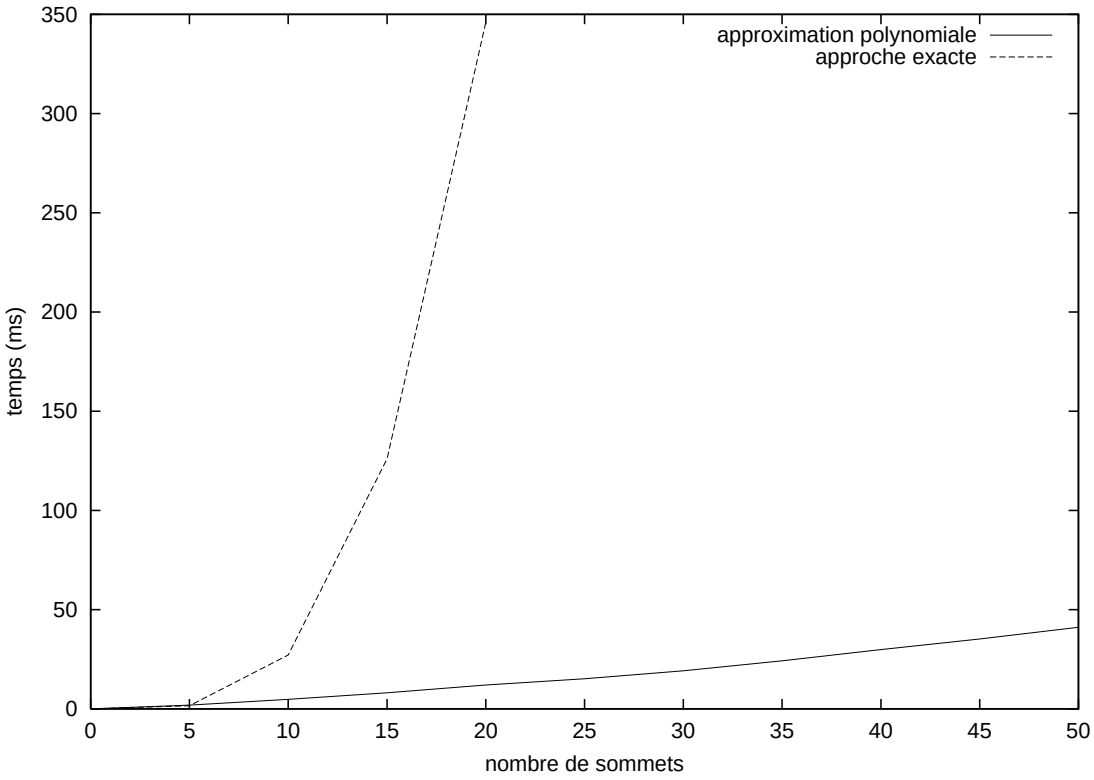


FIG. 3.17 – Temps d'exécution des deux approches de minimisation.

les graphes) nous avons obtenu des temps d'exécution similaires, avec un « décrochage » de PIP entre 20 et 30 sommets.

3.6 Conclusion

Dans ce chapitre, nous avons pu apporter des réponses précises concernant des points jusqu'alors inconnus de la technique du réordonnancement par niveaux. Tout d'abord, nous avons reformulé l'expression du coût mémoire d'une boucle, aussi appelé *maxlive*, d'une façon proche de celle de Ning et Gao (voir [82]). Ce coût donne une évaluation très précise du nombre de registres requis pour l'exécution de la boucle, comme le montrent [37, 71, 95]. De plus, nous avons montré qu'une fois l'ordonnancement déterminé, contrairement à ce que pourrait laisser penser une étude trop rapide du problème, l'évaluation de ce coût ne dépend pas de la taille du motif de l'ordonnancement cyclique, mais du nombre d'opérations de la boucle : il en découle que les techniques d'optimisation du coût mémoire d'une boucle (et entre autres le réordonnancement par niveaux) ne sont pas intrinsèquement pseudo-polynomiales[6].

Un problème comme la maximisation du partage des registres présenté par Leiserson et Saxe dans [70] possède une formulation relativement proche du problème du réordonnancement par ni-

[6] En effet, la plupart des formulations précédentes considèrent la coût mémoire d'une boucle comme le maximum du coût sur l'ensemble des cycles du motif de l'ordonnancement, autrement dit pour $0 \leq t < \lambda$. C'est le cas par exemple des formulations présentées dans [82, 35]. La conséquence directe d'une telle évaluation est que tout algorithme de minimisation du coût mémoire ne peut être que d'une complexité au mieux proportionelle à λ, donc pseudo-polynomiale.

veaux, et sa résolution est polynomiale. Un autre problème, associant une approximation de *maxlive* à un ordonnancement en ressources infinies, est résolu par Ning et Gao [82] par un algorithme polynomial. Ces résultats nous laissaient donc espérer un algorithme polynomial pour la résolution de notre problème. Pourtant, contre toute attente, nous avons pu prouver que le réordonnancement par niveaux est un problème fortement NP-complet. Il semble que l'aspect combinatoire du problème provienne de l'expression du coût mémoire d'une solution dont l'évolution est difficilement prévisible (dans la preuve de NP-complétude, nous mettons en place deux structures dont le coût diminue pour l'une et augmente pour l'autre pour un même *retiming*). Nous avons alors proposé une formulation exacte du problème par un programme linéaire en nombre entiers possédant un nombre polynomial de contraintes. Cette solution améliore la seule solution exacte qu'il est possible de trouver dans [36]. En effet, la solution proposée dans [36] est fondée sur la résolution d'un nombre exponentiel de programmes linéaires polynomiaux, mais possédant chacun un nombre exponentiel de contraintes. Nous avons montré que cette différence est due, entre autres, à un mauvais choix de variables pour l'expression du problème.

Nous avons implanté notre programme linéaire à l'intérieur de l'outil PASTAGA présenté en section 2.5 et nous avons pu constater qu'en pratique sa résolution est envisageable. Cependant, par souci de complétude, nous avons tenu à inclure dans ce chapitre un algorithme d'approximation du problème général. En éliminant la source de difficulté dans l'évaluation du coût en registres d'une boucle, et en nous rapprochant ainsi du problème de maximisation du partage des registres ainsi que du problème résolu par Ning et Gao, nous avons pu proposer un algorithme déterminant une solution approchée au problème du réordonnancement par niveaux dont la complexité est polynomiale. En outre, ceci nous a permis de confirmer que la difficulté du problème provient effectivement de l'expression complexe du coût.

De plus, comme l'ont démontré Eichenberger, Davidson et Abraham dans [33, 36], sur un grand nombre d'expérimentations le gain en registres produit par le réordonnancement par étages sur des programmes réels peut être très important et semble justifier même un algorithme de complexité exponentielle. Notre solution exacte d'une part et notre approximation d'autre part permettent de s'adapter aux situations où la consommation mémoire est critique pour l'une comme aux situations où la vitesse de compilation est prépondérante pour l'autre. Dans tous les cas, quelle que soit la méthode choisie, l'avantage du réordonnancement par étages est qu'il ne touche pas au motif de l'ordonnancement cyclique, donc les conflits de ressources restent satisfaits et la performance inchangée. Autrement dit, cette technique ne peut qu'améliorer la qualité d'un ordonnancement (ou au pire le laisse inchangé), son usage ne peut donc être que bénéfique.

Enfin, initialement notre étude de cette technique a été motivée par la forte augmentation de la consommation en registres des boucles sur lesquelles nous appliquions nos algorithmes de pipeline logiciel (voir section 2.5). Bien que particulièrement adapté à cette situation, nous devons remarquer que le réordonnancement par étages est une technique plus générale encore qui peut s'appliquer à toute boucle, même non pipelinée. En effet, il ne s'agit que d'un déplacement entre itérations des instructions, et le corps de la boucle ne joue ici aucun rôle excepté celui de limite à la légalité des déplacements possibles. Dans le cas d'une boucle séquentielle, il suffit juste de respecter la séquentialité des opérations. Ainsi, le réordonnancement par étages est bien une technique générale d'optimisation mémoire qui dépasse son cadre initial.

Chapitre 4

Parallélisation de boucles

4.1 Introduction

Depuis plus de vingt ans, les transformations de programmes produisant des boucles parallèles ont été intensément étudiées, aboutissant à un grand nombre d'algorithmes de complexité et d'efficacité variables (et plusieurs ouvrages sur le sujet, voir par exemple [109, 110]). Pourtant, souvent, dans les programmes traités, en particulier dans les programmes de calcul scientifique, la plupart des boucles sont déjà parallèles ou peuvent être parallélisées à l'aide de techniques extrêmement simples comme la distribution de boucles (qui est à la base de l'algorithme d'Allen, Callahan et Kennedy, voir [4]). Certains programmes dans lesquels l'interaction entre les calculs est plus forte – et dont le graphe de dépendance présente des circuits de récurrence – peuvent cependant nécessiter une transformation plus générale. Il est alors possible d'appliquer une torsion de boucle (*loop skewing*) comme dans la méthode des hyperplans de Lamport (voir [68]), ou une transformation unimodulaire plus générale (combinant la torsion, l'échange et l'inversion de boucles) comme dans l'algorithme de Wolf et Lam (voir [108]). Évidemment, ces deux dernières méthodes ont leurs limitations, puisqu'elles ne considèrent que les vecteurs de dépendance afin de transformer le domaine d'itération d'un nid de boucles : certains cas plus complexes pour lesquels la structure du graphe est importante peuvent avoir besoin de combiner une transformation linéaire à une distribution de boucle pour que des nids parallèles apparaissent. C'est ce que réalisent l'algorithme de Darte et Vivien (voir [27]) ou encore des méthodes plus générales comme l'algorithme de Feautrier ou celui de Lim et Lam (voir [41, 42, 74]). Mais plus la transformation réalisée par l'algorithme est générale, moins les nids de boucles pour lesquels elle est réellement nécessaire sont fréquents. Les transformations les plus complexes sont donc similaires à des outils chirurgicaux qui seraient la plupart du temps utilisés pour un travail de boucherie. Cette image caricaturale donne pourtant une représentation assez fidèle des situations réelles et à la question « pourquoi ne pas toujours utiliser les transformations les plus générales puisqu'elles intègrent les plus simples ? » nous pouvons avancer les arguments suivants :

- la complexité des algorithmes de transformation augmente avec leur généralité. Dans un compilateur, il n'est pas inutile de se demander si le résultat d'une transformation nécessaire pour une infime fraction des programmes vaut le prix d'un temps de compilation globalement élevé. C'est particulièrement vrai dans les compilateurs classiques destinés à un usage courant. Bien sûr, ceci ne rend pas les transformations les plus générales inutiles : elles sont d'une part nécessaires afin de comprendre le problème de la parallélisation d'un point de vue général, et d'autre part utiles lorsque le temps de compilation a une faible importance comparé à la qualité requise pour le résultat. Cette dernière situation se retrouve par exemple dans le

compilateur PICO développé aux HP Labs [98]. Il s'agit d'un compilateur de circuits pour systèmes embarqués, pour lequel la qualité du circuit produit importe avant tout ;

- plus la solution donnée par un algorithme est générale, moins il est possible de contrôler sa forme finale : une transformation simple, si elle est possible, pourra très bien être manquée ou tout simplement ignorée. Par exemple, certains algorithmes, fondés sur un réordonnancement des itérations, cherchent du parallélisme en essayant de « porter » toutes les dépendances. Ils produiront systématiquement un code complètement distribué même à partir d'un nid parfait qu'il serait possible de paralléliser tel quel, simplement en autorisant la présence de dépendances indépendantes du nid de boucles ;
- le code généré par une transformation complexe est généralement lui aussi complexe. Le problème avec un code complexe est qu'un certain nombre d'optimisations ultérieures peuvent devenir impossibles si le code prend une forme trop difficile à analyser. Par exemple, dans un langage comme HPF (voir [66]) pour une plate-forme à mémoire distribuée, appliquer une transformation comme une torsion de boucle fait perdre de la régularité aux communications et peut rendre impossible une génération efficace de celles-ci. En outre, un code complexe introduit souvent des calculs d'indices comportant des opérations coûteuses comme des multiplications ou des modulos. Dans un contexte de compilation de circuits, ceci n'est pas seulement coûteux en temps de calcul, mais aussi en matériel (voir [20]). Et cette complexité du code généré n'est pas toujours due à l'algorithme de génération du code, mais bien à la transformation elle-même (voir [64, 11, 16]) ;
- des optimisations plus fines sont souvent difficiles à intégrer aux transformations les plus générales. En ce sens, les algorithmes les plus généraux n'intègrent pas réellement les algorithmes les plus simples, puisqu'un algorithme simple pourra souvent être étendu afin de combiner plusieurs optimisations (alors qu'une telle extension ne sera pas nécessairement applicable au cas général). Ici encore, nous pouvons prendre l'exemple des algorithmes fondés sur l'ordonnancement des itérations d'un nid de boucles, qui produisent du parallélisme en faisant porter toutes les dépendances par le nid : de façon surprenante, ils n'envisagent pas les codes comportant un maximum de réutilisation des données à l'intérieur du corps (dépendances indépendantes du nid) et donc une meilleure localité. Un autre exemple est celui du placement des données, le placement lui-même est un problème difficile (voir [30, 43]). Une transformation complexe peut rendre plus difficile encore la recherche ultérieure d'un bon placement.

Parmi toutes ces transformations révélant du parallélisme dans les nids de boucles, seulement deux peuvent être considérées comme « simples » dans le sens où elles ne changent pas la structure du domaine d'itération : ce sont la distribution et l'alignement de boucle. La distribution de boucle réplique la structure de la boucle en plusieurs boucles successives sans changer le domaine d'itération. Cette transformation est l'outil de base de l'algorithme d'Allen, Callahan et Kennedy [4] ; en pratique elle est suffisante pour la détection de parallélisme dans de nombreux cas. Sa simplicité permet aisément de la combiner à d'autres objectifs : la distribution partielle [21] peut être utilisée pour produire des boucles parallèles contenant des dépendances indépendantes de la boucle, les techniques de fusion [77] permettent d'obtenir une meilleur localité une fois le code distribué, la distribution peut être étendue au cas de programmes à flot de contrôle complexe [65], etc.

Notre but dans ce chapitre est d'étudier de façon similaire l'alignement de boucle. L'alignement de boucle est fondé sur le décalage d'instructions, et a pour but de paralléliser les boucles internes d'un nid. La question qui nous intéresse ici est la suivante : pouvons-nous déterminer dans quelles circonstances un simple décalage est suffisant pour révéler des boucles parallèles dans un nid ? De façon plus générale, cette question correspond également au problème de savoir si une partie linéaire donnée d'un ordonnancement peut être complétée par un décalage adéquat produisant du parallé-

lisme. Cette question prend une importance toute particulière dans le cadre de certains algorithmes de compilation, où la parallélisation est réalisée par essais successifs de différentes transformations linéaires, et choix de la meilleure (voir [63, 62]). Dans ce cas, compléter la transformation linéaire par un décalage adéquat pourrait élargir l'ensemble des transformations possibles afin de trouver une solution encore meilleure. Ce problème de décalage que nous souhaitons étudier correspond en fait au choix de constantes de décalage dans certains algorithmes de parallélisation (voir [42, 74]).

Récemment, l'idée d'utiliser le décalage seul dans la parallélisation de boucles a été étudiée par Okuda [83], qui remarqua qu'en autorisant les dépendances indépendantes de la boucle il était finalement possible d'obtenir un code comportant moins de synchronisations. Il suggère donc dans son article de décaler les instructions afin de transformer certaines dépendances portées par la boucle en dépendances indépendantes de la boucle. Auparavant, la technique de l'alignement, qui repose sur la même idée, avait été proposée par Peir [89] (voir également [109] pour une présentation différente de cette technique). Cependant, aucun d'entre eux n'a alors répondu complètement à la question posée précédemment. L'idée plus générale de voir la parallélisation de boucle comme la combinaison d'une partie linéaire et d'un décalage a été développée dans [26], mais là encore les cas d'existence de ce décalage n'ont pas pu être caractérisés. Enfin, dans [69], notre problème est traité : un algorithme polynomial répondant à la question est donné, mais dans ce chapitre nous arrivons à une conclusion différente en prouvant que le problème est fortement NP-complet.

Le reste de ce chapitre est organisé de la manière suivante. Tout d'abord, en section 4.2, nous donnons plusieurs exemples de boucles parallélisables par décalage. En section 4.3, nous présentons le problème de la parallélisation utilisant un décalage directement à l'intérieur des boucles que nous souhaitons paralléliser : c'est la technique d'alignement sous sa forme classique, que nous appelons également *retiming* interne. Ce problème peut être résolu en temps polynomial, mais les conditions d'existence d'une solution sont très fortes et peu de cas peuvent être parallélisés de cette manière. Nous montrons alors en section 4.4 comment un décalage sur des boucles englobantes, que nous appelons *retiming* externe, peut rendre possible la parallélisation par *retiming* interne. Nous montrons alors que notre problème général peut être formulé en termes de *retimings* interne et externe et que sa résolution est un problème NP-complet au sens fort. Une solution exacte peut toutefois être exprimée sous forme d'un nombre polynomial de contraintes linéaires en nombres entiers. En section 4.5, nous donnons également des moyens pratiques de résolution du problème adaptés à divers cas de figure, et pouvant très facilement être intégrés à l'intérieur d'un compilateur paralléliseur. Enfin, en section 4.6, nous concluons sur ce chapitre.

4.2 Exemples, problèmes

Le but dans cette section est de montrer quels sont les problèmes de décalage que nous nous posons et leur lien avec d'autres approches et d'autres objectifs. Nous souhaitons surtout montrer où se situe le manque dans la compréhension de cette technique et insister sur les aspects complexes de son utilisation.

4.2.1 Alignement de boucle

Avant tout, nous nous devons de présenter la technique de l'alignement de boucle, la première, à notre connaissance, à mettre en œuvre le décalage d'instructions dans le cadre de la parallélisation de boucles. Nous le faisons sur un exemple, avant d'y revenir plus formellement à la section 4.3.

Exemple 1

Considérons l'exemple bien connu de [90] (cet exemple peut être retrouvé dans de nombreux articles du domaine) et son graphe de dépendance associé en figure 4.1. Afin de trouver du parallélisme sur cet exemple, Peir et Cytron utilisent un alignement multi-dimensionnel. L'idée de la technique est de grouper sur un seul arc le poids de toutes les dépendances formées par un circuit dans le graphe, en décalant les instructions les unes par rapport aux autres. En groupant ainsi les dépendances, il est possible que chaque circuit ne contienne plus que des dépendances indépendantes de la boucle, excepté une seule portée par la boucle (celle sur laquelle les poids ont été groupés).

Do $i = -1, 13$
  Do $j = -4, 25$
    S1 : $a(i,j) = b(i, j-6) + d(i-1, j+3)$
    S2 : $b(i+1, j-1) = c(i+2, j+5)$
    S3 : $c(i+3, j-1) = a(i, j-2)$
    S4 : $d(i, j-1) = a(i, j-1)$
  Enddo
Enddo

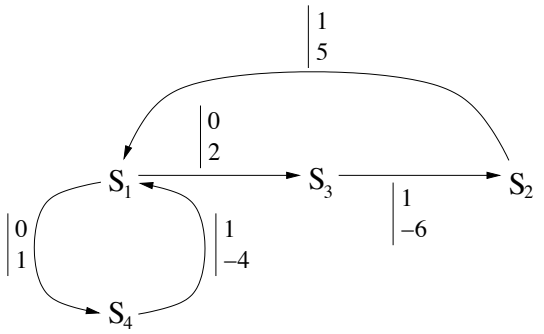
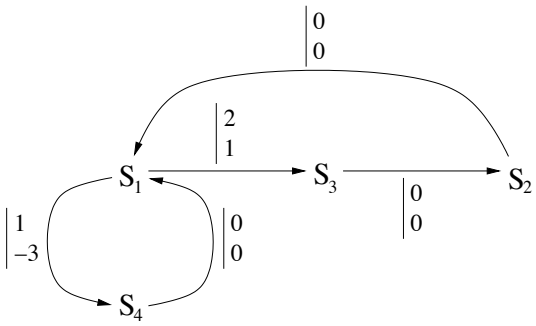


FIG. 4.1 – Un exemple tiré d'un article de Peir et Cytron.

Ici, l'alignement de boucle permet de trouver le code présenté en figure 4.2[1], dans lequel la boucle externe est séquentielle tandis que la boucle interne est parallèle. L'instruction $S4$ est décalée de $(1, -4)$ itérations (soit 1 itération sur la boucle externe et -4 sur la boucle interne), $S3$ et $S2$ respectivement de $(2, -1)$ et $(1, 5)$ itérations, et l'ensemble des instructions est réordonnancé dans le corps du nid de boucles suivant les dépendances indépendantes du nid obtenues après alignement. Ainsi, l'alignement résout notre problème sur cet exemple (bien que cela ne soit pas le cas en général).



Do $i = -1, 15$
  Doall $j = -8, 30$
    if $(i-1 \in [-1;13])$ and $(j+4 \in [-4;25])$ then
      S4 : $d(i-1, j+3) = a(i-1, j+3)$
    if $(i-2 \in [-1;13])$ and $(j+1 \in [-4;25])$ then
      S3 : $c(i+1, j) = a(i-2, j-1)$
    if $(i-1 \in [-1;13])$ and $(j-5 \in [-4;25])$ then
      S2 : $b(i, j-6) = c(i+1, j)$
    if $(i \in [-1;13])$ and $(j \in [-4;25])$ then
      S1 : $a(i,j) = b(i, j-6) + d(i-1, j+3)$
  Enddo
Enddo

FIG. 4.2 – La transformation proposée par Peir et Cytron (version « if »).

Néanmoins, pour ce même exemple, nous pouvons également utiliser une technique de décalage plus simple : la seule chose réellement nécessaire ici pour rendre la boucle parallèle est d'aligner

[1] Dans cette solution, nous avons exprimé le code sous la forme générale présentée par la définition 16 de la page 20 ; dans leur article [90], Peir et Cytron ne présentent que le cœur d'une forme avec prologue/épilogue (après épluchage), c'est pourquoi leur code peut sembler plus simple.

seulement les arcs non portés par la boucle externe afin de les rendre indépendants du nid. Il n'est pas nécessaire de décaler dans les deux dimensions, ni dans de telles proportions : un simple décalage interne est suffisant. Comme nous pouvons le constater sur la figure 4.3, nous obtenons alors un code plus simple, dans lequel les tests n'ont besoin d'être réalisés que sur l'indice j (la dimension dans laquelle nous avons décalé), et comme la quantité de décalage est plus petite, le domaine d'itération du nid est lui aussi plus petit.

Do i=-1,13
  Doall j=-5,26
    if $(j-1 \in [-4;25])$ then
      S1 : $a(i,j-1) = b(i,j-7)+d(i-1,j+2)$
    if $(j \in [-4;25])$ then
      S2 : $b(i+1,j-1) = c(i+2,j+5)$
    if $(j+1 \in [-4;25])$ then
      S3 : $c(i+3,j) = a(i,j-1)$
    if $(j \in [-4;25])$ then
      S4 : $d(i,j-1) = a(i,j-1)$
  Enddo
Enddo

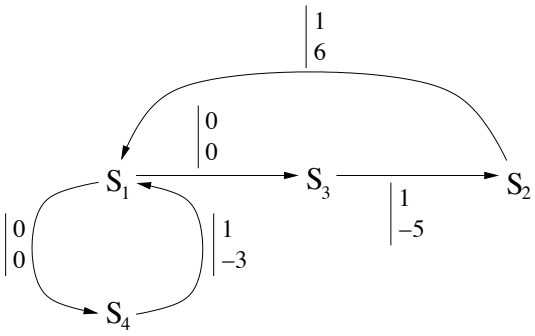
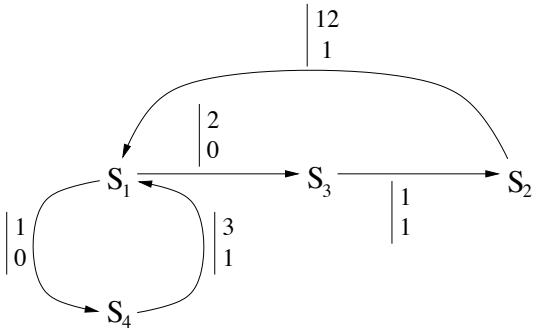


FIG. 4.3 – Notre solution pour l'exemple 1.

Comme le montre cet exemple, il est possible de trouver une boucle parallèle dans un nid simplement par l'application d'un décalage d'instructions dans une seule dimension. Nous pouvons légitimement nous demander « quand cela est-il possible ? » et « comment peut-on trouver le décalage approprié ? ». Tout vient à point à qui sait attendre, et nous tâcherons de répondre à ces deux questions par la suite. Notons que pour cet exemple nous aurions également pu utiliser une torsion de boucle pour trouver une boucle parallèle, et nous aurions obtenu la solution totalement différente présentée en figure 4.4 (dans ce cas le facteur de torsion minimum est 7 à cause de la dépendance de poids $(1,-6)$). Nous pouvons remarquer que dans cette solution nous n'avons pas



Do $j = -11, 116$
  Doall $i = \max(-1, \text{ceil}((j-25)/7)),$
    $\min(13, \text{floor}((j+4)/7))$
    S1 : $a(i, j-7\times i) = b(i, j-7\times i-6)$
      $+ d(i-1, j-7\times i+3)$
    S2 : $b(i+1, j-7\times i-1) = c(i+2, j-7\times i+5)$
    S3 : $c(i+3, j-7\times i-1) = a(i, j-7\times i-2)$
    S4 : $d(i, j-7\times i-1) = a(i, j-7\times i-1)$
  Enddo
Enddo

FIG. 4.4 – La solution la plus simple par torsion de boucle pour l'exemple 1.

besoin de « ifs », cependant cela ne signifie pas nécessairement que cette dernière est meilleure. En effet à chaque opération les anciens indices sont retrouvés à l'aide d'une multiplication, qui est une opération coûteuse, et parmi les autres problèmes nous retrouvons ceux que nous avions prévus : un grand nombre de pas séquentiels (127 au lieu de 31 pour notre solution), et une boucle interne dont les bornes varient au cours du temps.

4.2.2 Décalage d'instructions pour la fusion de boucles parallèles

Dans la section précédente, nous décalions les instructions d'un nid de boucles parfait. La situation générale peut s'avérer plus complexe. Par exemple, nous pouvons envisager de décaler des instructions appartenant à différentes boucles de façon à permettre leur fusion et à les grouper en un nombre minimal de boucles séquentielles et parallèles. Autrement dit, nous pouvons souhaiter combiner le décalage d'instruction à la fusion lors de notre recherche de boucles parallèles dans les nids parfaits ou non. Ce problème est étudié dans [9] et illustré sur l'exemple 2.

Exemple 2

Le code à gauche de la figure 4.5 peut être « compacté » sous la forme du code situé à droite de la figure (en supposant $N \geq 1$). Ceci minimise le nombre de boucles parallèles distinctes pouvant être obtenues par décalage et fusion.

```
loop1 :
  Doall i = 1, N
    a(i) = 1
  Enddo
loop2 :
  Doall i = 1, N
    b(i) = 1
  Enddo
loop3 :
  Doall i = 1, N
    c(i) = a(i-1) + b(i)
  Enddo
loop4 :
  Doall i = 1, N
    d(i) = a(i) + b(i-1)
  Enddo
loop5 :
  Doall i = 1, N
    e(i) = d(i) + d(i-1)
  Enddo
```

```
  a(1) = 1
  d(1) = a(1) + b(0)
loop1-2-4 :
  Doall i = 2, N
    a(i) = 1
    b(i-1) = 1
    d(i) = a(i) + b(i-1)
  Enddo
  b(n) = 1
loop3-5 :
  Doall i = 1, N
    c(i) = a(i-1) + b(i)
    e(i) = d(i) + d(i-1)
  Enddo
```

FIG. 4.5 – Décalage pour la fusion parallèle.

Grâce au décalage d'instructions, fusionner deux boucles est, théoriquement, toujours possible lorsque les dépendances sont uniformes : il suffit juste de décaler suffisamment pour que toutes les dépendances deviennent positives ou nulles (les boucles étant en séquence, le graphe est acyclique et nous retrouvons un simple problème de recherche de *retiming* légal). Cependant, lorsque nous souhaitons que le résultat soit une boucle parallèle, il faut trouver un décalage rendant nulles toutes les dépendances entre instructions à l'intérieur d'une même boucle : il ne doit plus rester que des dépendances indépendantes de la boucle, sinon la boucle devient soit incorrecte (dépendances négatives) soit séquentielle (dépendances positives). Dans cet exemple, une solution menant au nombre minimal de boucles parallèles est obtenue en décalant l'instruction de la boucle « loop2 » de 1.

Déterminer s'il existe un décalage d'une séquence de boucles de telle sorte que les boucles puissent être fusionnées en au plus K boucles parallèles distinctes est un problème NP-complet lorsque K est arbitraire ; la preuve se trouve dans [9]. En d'autres termes, décaler et fusionner un ensemble de boucles parallèles en aussi peu de boucles parallèles que possible est difficile, et ceci s'applique également aux nids de boucles. C'est pourquoi dans cette thèse nous nous restreignons à un cas plus simple, le cas où $K = 1$: quel que soit le nid de boucles que nous souhaitons paralléliser, nous cherchons un résultat sous la forme d'un nid de boucles parfait parallèle. Ceci signifie que si notre nid de boucles initial n'est pas parfait, nous allons réaliser sa fusion totale, et si nous souhaitons conserver la structure d'imbrication d'un nid de boucles non parfait, il nous suffira de nous intéresser à chaque sous-nid parfait indépendamment des autres. Cette fusion totale parallèle n'est pas NP-complète contrairement au cas de la fusion partielle. Nous y revenons plus en détails dans la section 4.3.1.

4.2.3 Décalage multi-dimensionnel

Déterminer si un décalage transformant une séquence de boucles en une unique boucle parallèle existe peut être réalisé efficacement (nous le montrons en section 4.3.1). Il suffit seulement de vérifier que le graphe formé des dépendances que nous souhaitons rendre nulles ne contient aucun cycle (non dirigé) de poids non nul. Le *retiming* adéquat se détermine alors en même temps que la vérification. Nous appelons cette transformation le *retiming* interne ; le problème est qu'il n'est pas toujours suffisant pour paralléliser une boucle.

Revenons alors au cas d'un nid de boucles : si nous ne sommes pas capables de paralléliser la boucle externe du nid, nous pouvons essayer de paralléliser la boucle interne. La différence ici est que les dépendances peuvent être portées par la boucle externe ou par la boucle interne que nous souhaitons paralléliser. Selon le cas, une dépendance apparaîtra ou non dans le graphe de dépendance de la boucle interne. Le *retiming* est une transformation qui modifie la distance de dépendance sur les arcs, donc un *retiming* de la boucle externe sera à même de modifier le graphe de dépendance de la boucle interne. Ainsi une nouvelle question se pose : est-il possible de trouver un *retiming* de la boucle externe garantissant qu'il existe un *retiming* de la boucle interne rendant cette dernière parallèle ? Autrement dit, si le graphe de dépendance de la boucle interne ne permet pas la parallélisation, peut-on tirer parti du *retiming* dans les deux dimensions afin de changer cette situation ?

Exemple 3

L'exemple 3 est une illustration du cas où un *retiming* externe bien choisi permet la parallélisation subséquente de la boucle interne. Dans le code de la figure 4.6, le calcul de $b(i, j)$ (instruction S2) dépend de deux valeurs distinctes du tableau a, calculées dans la même itération de la boucle externe mais dans deux itérations différentes de la boucle interne. Cette situation empêche la parallélisation de la boucle interne car le graphe à considérer pour cette boucle a alors deux arcs de S1 à S2 de poids respectifs 0 et 1, donc un cycle de poids non nul.

Après un *retiming* externe approprié, les deux dépendances qui nous gênaient deviennent portées par la boucle externe. La boucle interne est toujours séquentielle, car l'instruction S1 dépend de la valeur calculée par S2 dans l'itération précédente de la boucle interne. Mais cette fois, son graphe de dépendance ne contient pas de cycle. Ainsi, après un *retiming* interne, nous obtenons le code final 4.8 possédant une simple boucle parallèle interne.

```
Do i = 1, N
   Do j = 1, M
      S1 : a(i, j) = b(i − 1, j − 1)
      S2 : b(i, j) = a(i, j) + a(i, j − 1)
   Enddo
Enddo
```

FIG. 4.6 – Code initial de l'exemple 3.

```
Do i = 1, N + 1
   Do j = 1, M
      if (i ∈ [1, N]) then
         S1 : a(i, j) = b(i − 1, j − 1)
      if (i − 1 ∈ [1, N]) then
         S2 : b(i−1, j) = a(i−1, j)+a(i−1, j−1)
   Enddo
Enddo
```

FIG. 4.7 – Code après *retiming* externe.

```
Do i = 1, N + 1
   Doall j = 1, M + 1
      if (i − 1 ∈ [1, N]) and (j − 1 ∈ [1, M]) then
         S2 : b(i−1, j−1) = a(i−1, j−1)+a(i−1, j−2)
      if (i ∈ [1, N]) and (j ∈ [1, M]) then
         S1 : a(i, j) = b(i − 1, j − 1)
   Enddo
Enddo
```

FIG. 4.8 – Code final parallélisé.

Autrement dit, le but du *retiming* externe est de trouver un décalage permettant au graphe de dépendance de la boucle interne d'être parallélisable (c'est-à-dire de n'avoir aucun cycle de poids non nul). Nous prouvons en section 4.4.2 que ce problème est NP-complet au sens fort.

Cependant, afin de pouvoir malgré tout utiliser cette technique, nous pouvons tenter d'identifier des cas particuliers pour lesquels la résolution du problème est moins ardue et où une recherche partielle de solution suffit. La première solution qui vient à l'esprit est de s'arranger pour que le graphe de la boucle interne ne contienne aucun cycle, de cette manière il sera toujours parallélisable. Nous pouvons donc chercher un *retiming* externe transformant le graphe de dépendance de la boucle interne en forêt. Malheureusement, la preuve de NP-complétude construit un graphe à partir de 3-SAT tel que le seul moyen d'éliminer tous les cycles de poids non nul dans la dimension interne est également le seul moyen d'obtenir une forêt ou même un ensemble de chaînes. Ceci prouve donc dans le même temps que trouver un *retiming* de la boucle externe tel que le graphe résultant pour la boucle interne soit une forêt ou même un ensemble de chaînes est également NP-complet.

Le seul sous-problème restant que nous pouvons envisager – et qui soit plus simple – est de déterminer un *retiming* externe tel que le graphe de dépendance de la boucle interne ne contienne que des arcs de poids nul. Ce dernier problème est enfin soluble efficacement ; nous y reviendrons plus en détails en section 4.5 où nous envisagerons également d'autres moyens de résoudre le problème général.

4.3 Parallélisation par *retiming* interne

Dans cette section, nous présentons le cas d'un graphe de dépendance (pouvant correspondre à un nid de boucles parfait ou non) devant être transformé en un nid de boucles parfait totalement parallèle. Pour une simple boucle, cela revient à la rendre parallèle, et pour un nid cela revient à rendre toutes ses boucles parallèles. Pour ceci, nous cherchons un *retiming* capable de rendre

le poids de toutes les dépendances nul, ce qui correspond à des dépendances indépendantes de la boucle. Ce problème est déjà connu pour le cas d'une simple boucle[2] sous le nom d'alignement de boucle [109], mais la solution que nous proposons ici permet d'unifier ce problème avec les problèmes de *retiming* qui vont suivre (en particulier il fournit une base au cas multi-dimensionnel). En outre, notre solution est d'une complexité moindre que celle de l'algorithme suggéré dans [109, p. 404].

Par souci de clarté, nous commençons par présenter séparément le cas d'une boucle simple (graphe mono-dimensionnel), bien que le cas général soit essentiellement identique.

4.3.1 Cas mono-dimensionnel

Dans cette section, notre objectif est de formuler précisément les conditions sous lesquelles il existe un *retiming* r d'un graphe G tel que G_r n'ait que des arcs de poids nul. Ces résultats peuvent être retrouvés dans [23] et [89], nous en donnons ici une synthèse reformulée à l'aide de *retiming*. Nous commençons par un résultat intermédiaire établissant une condition suffisante à l'existence d'un *retiming* rendant nul le poids de tous les arcs du graphe.

Lemme 6 *Soit G un graphe de dépendance. Si G ne possède pas de cycle, alors il existe un* retiming *r tel que G_r n'ait que des arcs de poids nul.*

Preuve La preuve est une preuve constructive découlant directement de la structure du graphe. Nous choisissons un sommet arbitraire v_n pour chaque composante connexe n du graphe G et nous fixons $r(v_n) = 0$. Nous procédons alors à un parcours de chaque composante en partant de v_n et en suivant les arcs dans les deux directions. A chaque nouvel arc $e = (u, v)$ traversé, nous choisissons la valeur manquante $r(u)$ ou $r(v)$ (dépendant du sens dans lequel nous parcourons l'arc) de telle sorte que $r(v) - r(u) + d(e) = 0$. Puisque la valeur de *retiming* du sommet duquel nous venons est fixée (par hypothèse d'induction de la procédure de marquage), à chaque étape le choix d'une valeur est unique (déterminé par l'équation). Et puisque le graphe ne contient aucun cycle, chaque sommet n'est atteint qu'une seule fois, la valeur choisie n'est donc pas ambiguë et rend nul le poids de l'arc duquel nous venons. Enfin, puisque nous appliquons la procédure à chaque composante connexe, tous les arcs sont visités, il ont donc tous un poids nul après *retiming*. □

Munis de ce résultat intermédiaire, nous pouvons donner la condition nécessaire et suffisante à l'existence du *retiming* cherché dans le cas général.

Théorème 4 *Soit G un graphe de dépendance. Il existe un* retiming *r tel que G_r n'ait que des arcs de poids nul si et seulement si tous les cycles de G ont un poids nul.*

Preuve La nécessité de la condition est évidente : si, pour un *retiming* r, G_r ne contient que des arcs de poids nul alors pour tout cycle c de G_r, $d_r(c) = 0$ et, de par la proposition 1, $d(c) = 0$. G n'a donc que des cycles de poids nul.

Supposons maintenant que G n'ait que des cycles de poids nul. Soit T un arbre couvrant de G (en considérant G comme non dirigé). T n'a pas de cycle du tout (par définition, c'est un arbre), donc d'après le lemme 6, il existe un *retiming* r tel que T_r n'ait que des arcs de poids nul. Supposons alors que G_r ait un arc e de poids non nul. En ajoutant e à T nous formons un cycle c (autrement e aurait pu faire partie de T, et T ne serait pas un arbre couvrant). Ce cycle a un poids non nul puisqu'un de ses arcs est l'arc e de poids non nul après *retiming* par r, et que tous les autres sont des arcs de poids nul dans T_r, autrement dit $d_r(c) \neq 0$. Donc, par la proposition 1, $d(c) = d_r(c) \neq 0$:

[2] Mais nous pouvons noter que le cas d'un nid de boucles ne présente aucune différence du moment que nous disposons de l'addition, de la soustraction et d'un ordre sur les vecteurs de distance.

G possède un cycle de poids non nul, ce qui contredit l'hypothèse de départ. Ainsi tous les arcs de G_r ne peuvent avoir qu'un poids nul ce qui prouve que la condition est suffisante. □

Le théorème 4 montre qu'il n'est pas suffisant qu'un graphe n'ait pas de circuits de poids nul pour être parallélisable, mais qu'il faut que même les cycles du graphe aient tous un poids nul. Ceci va à l'encontre d'une intuition qui pourrait laisser croire qu'un graphe acyclique (donc sans circuit) peut toujours être parallélisé par *retiming*. C'est cette confusion qui a été commise dans [69] et [26], et grâce à laquelle nous pouvons trouver dans [69] un algorithme polynomial pour résoudre le problème plus général de parallélisation dans le cas multi-dimensionnel (voir section 4.4). Évidemment cette intuition est fausse, et un graphe sans circuit peut très bien avoir un cycle de poids non nul (par exemple, si une paire de sommets possède deux chemins de poids différents les reliant). Pourtant, le résultat du théorème 4 n'est pas nouveau, Peir l'avait déjà prouvé dans [89, *Lemma* 4.1].

L'algorithme suggéré dans [109] pour transformer par *retiming* une séquence de boucles simples – dont les dépendances sont uniformes – en une simple boucle parallèle, vérifie la propriété du théorème 4 en transformant tous les cycles en circuits, et en utilisant un algorithme de Bellman-Ford. Tout d'abord, un nouveau graphe est construit en ajoutant, pour chaque arc $e = (u, v)$ du graphe G, un nouvel arc $e' = (v, u)$ de poids $d(e') = -d(e)$. Cette transformation permet de simuler le parcours de l'arc en sens contraire à sa direction (le poids d'un arc parcouru en sens inverse est soustrait dans le calcul du poids d'un cycle). Ainsi, après la transformation, pour tout cycle du graphe original, il existe deux circuits dans le graphe transformé correspondant au parcours du cycle dans les deux sens possibles. Si le cycle a un poids non nul, alors l'un des deux circuits sera de poids négatif. Il est donc possible à l'aide d'un algorithme de Bellman-Ford de savoir si le *retiming* cherché existe et, si c'est le cas, le plus court chemin calculé nous fournit également les valeurs de *retiming*. La complexité d'un tel algorithme est en $O(|V||E|)$.

En fait, il n'est pas nécessaire de mettre en œuvre une procédure aussi lourde : la preuve du théorème 4 nous suggère un algorithme beaucoup plus simple et moins complexe. Cet algorithme est fondé sur une traversée d'un arbre couvrant de G. Nous supposons G connexe, autrement la procédure est à appliquer à chaque composante connexe.

Algorithme 11 (*Parallélisation par* retiming *interne*)

- pour chaque sommet v de G, poser $m(v) = 0$;
- poser $S = \{v_0\}$ où v est un sommet arbitraire de G, poser $m(v_0) = 1$ et $r(v_0) = 0$;
- tant que S n'est pas vide, choisir un sommet $v \in S$, poser $S = S - \{v\}$, et :
  - pour chaque arc $e = (u, v)$ vérifiant $m(u) = 0$, poser $r(u) = r(v) + d(e)$, $m(u) = 1$, et $S = S \cup \{u\}$;
  - pour chaque arc $e = (v, t)$ vérifiant $m(t) = 0$, poser $r(t) = r(v) - d(e)$, $m(t) = 1$, et $S = S \cup \{t\}$.
- si tous les arcs de G_r sont nuls, alors retourner VRAI, sinon retourner FAUX.

Les initialisations de cet algorithme sont en $O(|V|)$, la procédure de marquage en $O(|V| + |E|)$ et la vérification du poids de tous les arcs en $O(|E|)$. Donc la complexité totale est en $O(|V| + |E|)$. Notons que ce simple algorithme n'est rien d'autre que l'algorithme 4.1 de [89].

Exemple 4

Nous proposons ici un programme fabriqué dans le but d'illustrer notre technique. Il est donné en figure 4.9 avec son graphe de dépendance, qui est sans cycles de poids non nul.

Do $i = 1, N$
  $a(i) = d(i-1) * d(i-1) + b(i-1)$
  $b(i) = sin(h(i))$
  $c(i) = cos(h(i))$
  $d(i) = b(i) + c(i-1)$
  $e(i) = sin(c(i))$
Enddo
Do i=1,N
  $f(i) = a(i+1) + e(i-1)$
  $g(i) = f(i) * a(i+1)$
Enddo

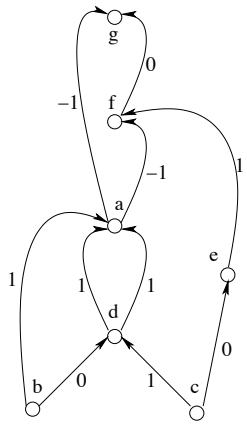
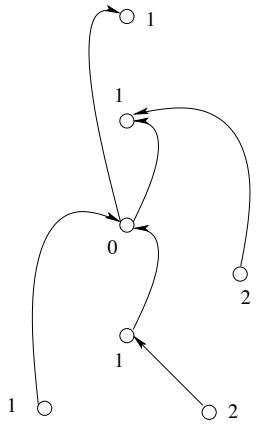


FIG. 4.9 – Exemple de programme parallélisable.

Un arbre couvrant du graphe et un *retiming* associé sont donnés en figure 4.10. Sur cet exemple, le *retiming* trouvé produit un code parallèle. Une autre solution aurait pu être trouvée selon l'arbre couvrant et le sommet initial choisis, cependant toutes les solutions diffèrent uniquement par une constante (la preuve est immédiate, il suffit de remarquer que $r(v) - r(u) = d(e)$ et $d(e)$ est fixe). Notons que les instructions ont du être réordonnées afin de respecter les nouvelles dépendances de poids nul. Par souci de simplicité, nous avons omis le code correspondant à l'épilogue et au prologue.



prologue
Doall $i = 3, N$
  $b(i-1) = sin(h(i-1))$
  $c(i-2) = cos(h(i-2))$
  $d(i-1) = b(i-1) + c(i-2)$
  $a(i) = d(i-1) * d(i-1) + b(i-1)$
  $e(i-2) = sin(c(i-2))$
  $f(i-1) = a(i) + e(i-2)$
  $g(i-1) = f(i-1) * a(i)$
Enddo
épilogue

FIG. 4.10 – Arbre couvrant choisi et programme parallélisé pour l'exemple de la figure 4.9.

4.3.2 Extension aux nids de boucles

Dans cette section, nous étendons l'algorithme 11 de parallélisation pour une simple boucle au cas multi-dimensionnel. Il y a deux cas à envisager pour cette extension : le cas de la parallélisation totale d'un nid (toutes les boucles deviennent parallèles) et le cas de la parallélisation des x boucles les plus externes dans le nid final.

Le premier problème est très similaire au problème mono-dimensionnel d'alignement d'une simple boucle que nous avons étudié. En fait, tous les résultats précédents peuvent être utilisés dimension par dimension (car un cycle a un poids nul si et seulement si son poids est nul dans chacune des dimensions), mais la meilleure méthode est encore de considérer directement toutes les dimensions en utilisant des vecteurs au lieu de scalaires dans notre algorithme. Ainsi, il existe un *retiming* multi-

dimensionnel rendant le poids de tous les arcs nul si et seulement si le graphe de dépendance du nid contient seulement des cycles de poids nul. L'algorithme fonctionne toujours moyennant l'hypothèse que nous disposons d'une opération d'addition, d'une opération de soustraction et d'un ordre total sur les vecteurs de dépendance (ce qui est généralement le cas).

Cependant, une légère complication apparaît pour le second problème. Nous souhaitons obtenir un nid parfait de profondeur p dont les x premières boucles sont parallèles. Le problème est que si nous nous contentons d'appliquer l'algorithme d'alignement sur les boucles qui nous intéressent (c'est-à-dire les x premières dimensions), nous risquons de voir apparaître un arc de poids lexico-négatif, auparavant porté par une des boucles externes que nous venons de paralléliser. Autrement dit, si certains vecteurs de dépendance ont des composantes négatives au-delà des x premières, nous devons compléter la transformation à l'aide d'un *retiming* des dernières dimensions (ou d'une distribution) de telle sorte que le résultat final reste correct. Cette situation est illustrée par la figure 4.11, pour laquelle la parallélisation de la boucle externe est réalisée en deux temps : la parallélisation proprement dite et la correction des poids dans la boucle interne. Heureusement, tout graphe produit par *retiming* à partir d'un graphe correct est lui aussi correct, simplement parce que le corollaire 1 nous assure que le poids d'un circuit est inchangé par *retiming*. La correction des poids après parallélisation des boucles externes est donc toujours possible. Une fois les x premières boucles parallélisées, tous les vecteurs de dépendance ont leurs x premières composantes nulles. Le graphe des boucles internes possède donc les mêmes arcs et les mêmes circuits. De plus, le poids chacun de ces circuits est toujours lexico-positif. Le graphe des boucles internes est donc correct, il n'a pas de circuits de poids lexico-négatif ou nul. Nous pouvons donc, grâce à l'algorithme 2, trouver un *retiming* légal de ce graphe et produire un code correct. Pour ce *retiming* légal, nous n'avons pas à nous soucier de problèmes de parallélisme, les boucles internes peuvent être séquentielles.

Cette remarque nous permet donc de garantir que nous pouvons corriger tout nid après parallélisation de sa boucle externe. Ainsi, si nous appliquons l'algorithme d'alignement boucle après boucle en partant de la plus externe et en corrigeant le code final dès que l'algorithme échoue, nous obtenons un moyen simple et efficace de trouver le nombre maximum de boucles parallèles externes par *retiming*. Dans une autre optique, il est possible de vérifier dimension par dimension si une boucle à un niveau donné est parallélisable. En appliquant ensuite un échange de boucles, nous nous retrouvons avec le nombre maximum de boucles parallèles externes qu'il est possible d'obtenir par combinaison de *retiming* et d'échange de boucles[3].

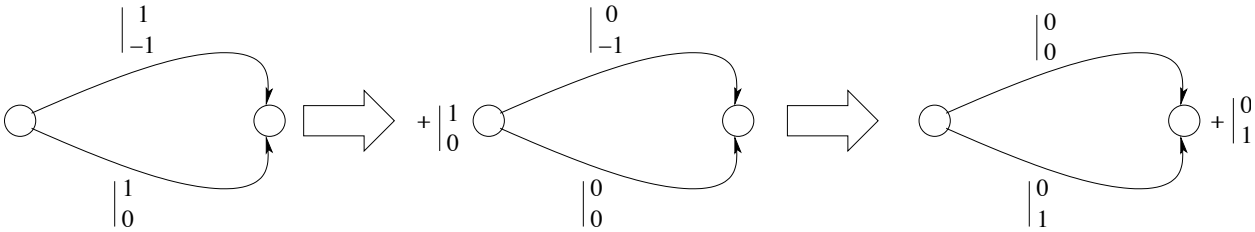


FIG. 4.11 – Parallélisation de la boucle externe seulement.

Ceci conclut cette section. La parallélisation par *retiming* interne est soluble efficacement. Elle consiste à utiliser un *retiming* uniquement le long des dimensions contenues dans les boucles que

[3]Dans ce cas également, nous pouvons garantir la correction. En effet, si nous commençons par rendre nuls les poids dans les dimensions parallélisables, nous vérifions toujours l'hypothèse de lexico-positivité des circuits. Nous pouvons alors passer les boucles parallèles correspondantes à l'extérieur du nid sans changer la correction du graphe, puisque toutes les dépendances ont une composante nulle pour ces boucles. Nous nous retrouvons alors dans le cas précédent de boucles parallèles toutes externes.

nous souhaitons paralléliser. Il suffit pour cela de vérifier que tous les cycles du graphe ont un poids nul, ce qui peut se faire par un simple parcours. Dans la section suivante, nous nous intéresserons au problème complémentaire du *retiming* externe. Il consiste à trouver un décalage d'une ou plusieurs boucles englobantes de telle sorte que la ou les boucles internes deviennent parallélisables par *retiming* interne. Comme nous le verrons, le problème s'avère plus complexe.

4.4 Parallélisation par *retiming* externe

Lorsque le *retiming* interne n'est pas suffisant pour paralléliser une boucle, nous pouvons tenter d'appliquer un *retiming* à une boucle externe (ou englobante) afin de transformer le graphe de dépendance de la boucle interne et en permettre la parallélisation. En effet, dans ce graphe de dépendance, les arcs portés par une boucle externe n'apparaissent pas. L'idée est donc de tenter d'exploiter à bon escient ces arcs pour éliminer toutes les contraintes empêchant la parallélisation. Grâce au théorème 4, nous savons que ces contraintes se résument aux cycles de poids non nul. Autrement dit, il suffit de faire en sorte qu'au moins un arc de chacun de ces cycles soit porté par la boucle externe pour résoudre notre problème de parallélisation. Le code résultant (en dimension 2) sera alors composé d'une boucle séquentielle entourant une boucle parallèle ; nous retrouvons la structure classique produite par les algorithmes de parallélisation fondés sur le réordonnancement des itérations. Ceci a été illustré par l'exemple 3 de la page 117.

Nous commençons par proposer une caractérisation globale de notre problème et nous montrons qu'il peut se réduire sans perte de généralité à une composition de sous-problèmes en dimension 2. Nous prouvons alors que le sous-problème en dimension 2 est NP-complet au sens fort, ce qui nous permet de conclure à la NP-complétude du cas général. Néanmoins, nous donnons une méthode de résolution exacte (bien qu'exponentielle) vraisemblablement applicable aux petits exemples.

4.4.1 Caractérisation du problème

Notre but ici est d'identifier de la façon la plus générale possible quels problèmes nous souhaitons résoudre lorsque le nid est d'une profondeur supérieure à 2. Nous considérons tout d'abord le problème de parallélisation en un nid parfait comportant un certain nombre de boucles séquentielles entourant un nid complètement parallèle. Soit un graphe de dépendance $G = (V, E, d)$ de dimension (profondeur) n pour lequel nous souhaitons trouver x boucles internes parallèles (donc entourées de $n - x$ boucles séquentielles), avec $1 \leq x < n$. Nous cherchons donc un *retiming* r tel qu'aucune des dépendances de G_r ne soit portée par une des x dernières boucles. Ceci se produit lorsque chaque dépendance est soit portée par l'une des $n - x$ premières boucles du nid, soit indépendante du nid. Plus formellement, nous dirons que les x boucles les plus internes sont parallèles lorsque :

$$\forall e = (u, v) \in E,\ d(e) \geq_{lex} (\underbrace{0, \ldots, 0, 1}_{n-x}, \underbrace{-\infty, \ldots, -\infty}_{x}) \text{ ou } d(e) = 0$$

Remarquons qu'avec cette définition, nous ne recherchons que des *retimings* légaux (ne produisant que des arcs de poids lexico-positifs). Grâce à l'algorithme 2, nous pouvons supposer que nous démarrons d'un graphe légal (ne possédant que des arcs de poids lexico-positifs). Si un *retiming* préalable est nécessaire pour arriver à ce graphe légal, il pourra être ajouté à la solution finale.

Nous commençons par montrer un résultat intermédiaire nous permettant de conclure que le *retiming* sur les boucles séquentielles peut être ramené à un *retiming* mono-dimensionnel (quel que soit le nombre des boucles séquentielles). Autrement dit, nous pouvons remplacer une combinaison de $n - x$ *retimings* externes par un seul produisant un résultat qualitativement équivalent. Dans ce

théorème, nous ne considérons que les boucles séquentielles. Nous montrons que, pour tout *retiming* multi-dimensionnel d'un nid, un *retiming* de la boucle la plus interne seulement peut porter au moins les mêmes arcs.

Théorème 5 *Soit G un graphe de dépendance légal de dimension y. Pour tout* retiming *légal r de G, il existe un* retiming *légal r' de la dimension y seulement tel que tous les arcs de poids strictement lexico-positif de G_r ont aussi un poids strictement lexico-positif dans $G_{r'}$.*

Preuve Considérons $G' = (V, E', d_y)$ le sous graphe de G obtenu en posant

$$E' = \{e \in E \mid d(e) <_{lex} (0, \ldots, 0, 1, -\infty)\}$$

autrement dit, E' est l'ensemble de tous les arcs non portés par l'une des $(y-1)$ premières boucles et où d_y est la restriction de d à sa $y^{\text{ème}}$ dimension. Soit

$$\forall e \in E',\ m(e) = \begin{cases} 0 & \text{si } d_r(e) = (0, \ldots, 0) \\ 1 & \text{sinon} \end{cases}$$

donc $m(e) = 0$ si e est indépendant de la boucle dans G_r (donc de poids nul après *retiming* par r) et $m(e) = 1$ sinon (donc si e est de poids strictement lexico-positif après *retiming* par r). S'il existe un *retiming* r' de G' tel que $\forall e \in E', d_{r'}(e)_y \geq m(e)$ alors r' est un *retiming* légal de la $y^{\text{ème}}$ dimension de G qui porte au moins les mêmes arcs que r : par définition de $m(e)$, les arcs de E' auront un poids positif et seront portés pour r' lorsqu'ils le sont pour r, et les arcs de $E \backslash E'$ ont un poids déjà strictement positif dans une dimension inférieure, par construction de E' et par légalité de G, donc ils restent portés et ne posent pas de problème.

D'après la preuve de l'algorithme 2, ceci est vrai si et seulement si le graphe $G' - m$ ne possède aucun circuit de poids strictement négatif. Tous les arcs de E' appartiennent aussi à G et ont un poids nul dans leurs $(y-1)$ premières dimensions (par construction). Tout circuit c de G' est donc un circuit de G formé uniquement d'arcs de E' ; il a un poids nul dans ses $(y-1)$ premières dimensions. C'est également un circuit de G_r dont le poids est également nul dans ses $(y-1)$ premières dimensions (par conservation du poids d'un circuit par *retiming*, corollaire 1). Par légalité de r, tous les arcs de c ont donc un poids dans G_r dont les $(y-1)$ premières dimensions sont nulles. Autrement dit, par définition de $m(e)$, chaque arc de c a un poids dans G_r strictement positif dans sa $y^{\text{ème}}$ dimension si $m(e) = 1$, et totalement nul si $m(e) = 0$. Donc le poids de c dans G_r est tel que $d_r(c)_y \geq m(c)$. Toujours par conservation du poids d'un circuit par *retiming*, nous en déduisons $d(c)_y - m(c) \geq 0$. Donc $G' - m$ ne possède aucun circuit de poids négatif. □

Pour reformuler ce théorème de façon plus intuitive, le *retiming* dans les $(y-1)$ premières dimensions ne peut rien changer aux circuits nuls dans leurs $(y-1)$ premières dimensions. Or, ces circuits sont les seules contraintes difficiles pour le *retiming* dans la $y^{\text{ème}}$ dimension. En effet, comme le montre l'algorithme 2, il n'est pas difficile de contraindre les poids à prendre une valeur arbitrairement grande par *retiming* dans un graphe sans circuits. Ainsi, un *retiming* multi-dimensionnel ou un *retiming* dans la dimension la plus interne seulement ont les mêmes contraintes lorsqu'il s'agit de porter les arcs.

Le théorème 5 montre que, sans perte de généralité, nous pouvons réduire notre problème initial à la recherche d'une simple boucle séquentielle entourant un nid complètement parallèle. En outre, comme nous avons pu le constater en section 4.3.2, paralléliser une simple boucle ou paralléliser complètement un nid est en fait le même problème. Dans les deux cas, la condition à satisfaire est que le graphe ne contienne pas de cycles de poids non nul. L'algorithme 11 de la page 120 s'applique

sans modification en remplaçant les valeurs de poids et de *retiming* scalaires par des vecteurs. En résumé, et sans perte de généralité, nous supposerons que notre problème de *retiming* d'un nid tel que les x boucles les plus internes soient parallèles se réduit à la recherche d'un *retiming* d'un nid de deux boucles imbriquées tel que la boucle interne soit parallèle.

Finalement, la proposition suivante sépare le problème en deux sous-problèmes. Elle nous permet de chercher de manière séparée le *retiming* dans la première dimension et celui dans la seconde. Nous commençons par introduire une notation qui devrait faciliter la compréhension.

Définition 23 (sous-graphe de la boucle interne) *Soit $G = (V, E, d)$, un graphe de dépendance. Nous notons $\tilde{G} = (V, \tilde{E}, \tilde{d})$ le graphe de la boucle interne, où $\tilde{E} = \{e \in E \mid d(e) <_{lex} (1, -\infty)\}$, et où $\tilde{d}$ est la restriction de d à sa seconde composante : $\forall e \in \tilde{E}, \tilde{d}(e) = d(e)_2$. Nous notons $\tilde{G}_r$ le graphe $\tilde{H}$ où $H = G_r$ (le* retiming *est donc appliqué avant d'enlever les arcs. Dans le cas contraire, nous utilisons la notation $(\tilde{G})_r$).*

La proposition suivante est alors une conséquence directe du théorème 4.

Proposition 14 *Soit un graphe G de dimension 2, il existe un* retiming *légal r de G tel que la boucle interne de G_r soit parallèle si et seulement s'il existe un* retiming *légal r_1 de G dans sa première dimension seulement tel que $\tilde{G}_{r_1}$ n'ait que des cycles de poids nul.*

Nous avons donc formellement séparé le problème en deux : le problème de *retiming* sur la dimension externe qui doit permettre de porter les « bonnes » dépendances, afin que la boucle interne devienne parallélisable, et le problème de *retiming* sur la dimension interne devant effectuer la parallélisation proprement dite. La figure 4.12 donne un exemple de cette parallélisation en deux temps. Sur cette figure, les arcs en pointillés sont les arcs portés par la boucle externe. Si nous considérons le graphe initial, nous remarquons que la boucle externe ne peut pas être parallélisée (à cause d'un circuit de poids non nul). De plus, la boucle interne telle quelle ne peut pas être parallélisée non plus, à cause d'un cycle de poids non nul dans $\tilde{G}$. Cependant, si nous choisissons un *retiming* adéquat dans la première dimension, nous pouvons changer $\tilde{G}$ de manière à le réduire à un seul arc. La boucle interne devient alors parallélisable.

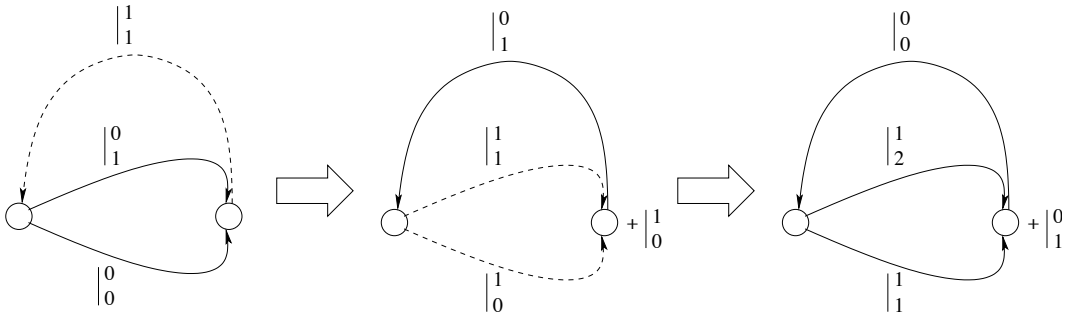


FIG. 4.12 – Une parallélisation en deux étapes de la boucle interne.

Même si pour cet exemple un *retiming* externe approprié était simple à trouver, nous montrons que dans le cas général le problème est NP-complet au sens fort.

4.4.2 NP-complétude

Le but de cette section est d'établir la NP-complétude du problème de parallélisation de la boucle interne en dimension 2. Par suite, le problème général de parallélisation d'une ou plusieurs boucles internes dans un nid plus large est également NP-complet. Notons que, sauf si P=NP, ceci va à

l'encontre du résultat de [69] où un algorithme polynomial est donné pour résoudre ce problème. Comme nous l'avons évoqué en section 4.3.1, l'erreur de [69] concerne les conditions sous lesquelles une boucle interne peut être parallélisée. À la section 4.4.3, nous revenons à l'algorithme de Passos et Sha et nous donnons un contre exemple à leur résultat. Mais tout d'abord, intéressons-nous à notre preuve de NP-complétude.

La preuve se fait par réduction polynomiale de 3SAT (problème LO2 de [47, p. 259]). Comme nous le verrons, nous construisons un graphe G à partir de 3SAT tel que si G est une instance positive de notre problème, alors pour toute solution r_1 (*retiming* dans la première dimension), $\tilde{G}_{r_1}$ est un ensemble de chaînes. Ceci prouve donc également, comme avancé en section 4.2, que la recherche d'un *retiming* r_1 de la première dimension tel que $\tilde{G}_{r_1}$ soit une forêt ou même un ensemble de chaînes est un sous-problème également NP-complet. La formulation de 3SAT est rappelée en section 3.4.1, nous pouvons donc passer directement à la formulation de notre problème. Grâce à la proposition 14, il peut être formulé de la façon suivante.

Problème : Parallélisation interne 2D

Instance Un graphe de dépendance $G = (V, E, d)$ de dimension 2.

Question Existe-t-il un *retiming* légal r de G dans sa première dimension seulement tel que $\tilde{G}_r$ n'ait que des cycles de poids nul ?

Tout d'abord, nous avons besoin d'un résultat intermédiaire qui nous permettra de prouver que le problème est dans NP. Le lemme suivant prouve que pour tout *retiming* du graphe, il existe un *retiming* portant les mêmes arcs dont les valeurs sont bornées.

Lemme 7 *Soit un graphe de dépendance $G = (V, E, d)$ de dimension 1 et un* retiming *légal r de G. Il existe un* retiming *légal r' de G tel que $\forall v \in V$, $0 \leq r'(v) \leq \sum_{e \in E} |d(e)| + |V|$ et $\forall e \in E, d_r(e) > 0 \iff d_{r'}(e) > 0$ (les arcs portés après* retiming *par r ou r' sont les mêmes).*

Preuve Nous construisons un graphe $G' = (V, E', d')$ à partir de G et r de la manière suivante :

- nous partons de $E' = E$;
- pour tout arc $e = (u, v) \in E$:
  - si e est tel que $d_r(e) = 0$, nous posons $d'(e) = d(e)$ et nous ajoutons à E' un arc $e' = (v, u)$ de poids $d'(e') = -d(e)$. Notons qu'alors e et e' forment un circuit de poids nul ;
  - sinon, $d_r(e) \geq 1$ et nous posons $d'(e) = d(e) - 1$.

Par construction, G'_r n'a que des arcs de poids positif ou nul. Donc r est légal pour G' et, par la conservation du poids d'un circuit par *retiming* (proposition 1), G' ne possède pas de circuit de poids négatif. Nous pouvons donc trouver à l'aide de l'algorithme 2 un nouveau *retiming* légal r' de G'. Par construction de G', ce nouveau *retiming* rend nuls exactement les mêmes arcs que r dans G. Ainsi, tous les arcs portés dans G_r sont également portés dans $G_{r'}$. De plus, l'algorithme 2 trouve pour chaque sommet u la valeur de *retiming* $r'(u)$ en prenant l'inverse du poids d'un plus court chemin d'un sommet quelconque vers u. Un tel plus court chemin a un poids négatif ou nul – car nous considérons qu'un chemin composé du seul sommet u a un poids nul (voir les détails de l'algorithme 2). Donc r' ne possède que des valeurs positives ou nulles. Et puisque le plus court chemin vers un sommet peut être défini par un chemin élémentaire dans le graphe, le nouveau *retiming* est tel que $\forall v \in V, 0 \leq r'(v) \leq \sum_{e \in E} |d'(e)| \leq \sum_{e \in E} |d(e)| + |V|$ (car au plus |V| sommets, donc |V| arcs, sont traversés par un chemin élémentaire). □

Nous sommes maintenant prêts à prouver que Parallélisation interne 2D est NP-complet au sens fort. Nous disons au sens fort car la réduction est faite à partir de 3SAT, problème n'impliquant pas de nombres, donc n'offrant pas de solution pseudo-polynomiale.

Théorème 6 *Le problème* Parallélisation interne 2d *est NP-complet au sens fort.*

Preuve Nous commençons par prouver que le problème appartient à NP, puis nous décrivons notre transformation polynomiale d'une instance de 3SAT en instance de Parallélisation interne 2d, enfin nous montrons l'équivalence entre instances positives.

Le problème est dans NP Considérons une instance positive de notre problème : un graphe $G = (V, E, d)$ de dimension 2, ainsi qu'une solution à cette instance, un *retiming* légal r de la première dimension de G uniquement tel que $\tilde{G}_r$ contienne seulement des cycles de poids nul. Grâce au lemme 7, nous savons qu'il existe un *retiming* r' qui, appliqué à G, mène aux mêmes arcs portés par la boucle que r et dont les valeurs sont bornées par une fonction polynomiale des poids de G dans la première dimension. Autrement dit, $\tilde{G}_r = \tilde{G}_{r'}$. Par conséquent, r' est un certificat polynomial de notre instance : sa taille est polynomiale en la taille de l'instance, et vérifier que $\tilde{G}_{r'}$ n'a aucun cycle de poids non nul peut être fait en temps polynomial par l'algorithme 11. Il en ressort que le problème appartient à NP.

Transformation Soit une instance (U, C) de 3SAT, nous définissons $f(U, C) = G = (V, E, d)$ comme une transformation polynomiale de (U, C) en instance de Parallélisation interne 2d.

Tout d'abord, nous décrivons une structure que nous utiliserons comme un composant de base de notre transformation ; nous appelons cette structure la *copie de sommet*. Nous construisons une *copie de sommet* y à partir d'un sommet x de G de la manière suivante (voir figure 4.13) :

- nous ajoutons à V un nouveau sommet y ;
- nous ajoutons à E deux arcs de x vers y de poids respectifs $(1, 0)$ et $(1, 1)$, et deux arcs de y vers x également de poids respectifs $(1, 0)$ et $(1, 1)$.

Munis de cette construction de base, nous construisons alors G comme suit :

- nous démarrons avec un sommet unique b_G : $V = \{b_G\}$, $E = \emptyset$;
- pour chaque variable $u \in U$, nous ajoutons une *copie de sommet* b_u de b_G, un nouveau sommet v_u, ainsi que deux arcs $e_u = (v_u, b_u)$ et $e_{\bar{u}} = (b_u, v_u)$ de poids respectifs $d(e_u) = (1, 0)$ et $d(e_{\bar{u}}) = (0, 0)$ (voir figure 4.14) ;

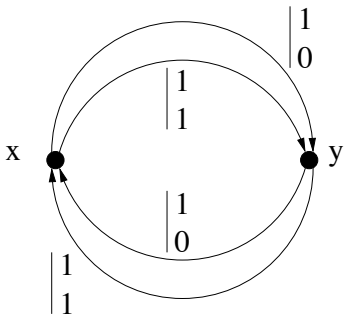


Fig. 4.13 – Structure de *copie de sommet*.

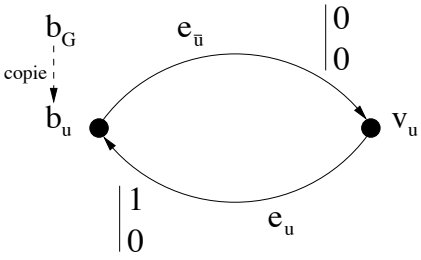


Fig. 4.14 – Transformation d'une variable u.

- pour chaque clause $c = \{x, y, z\} \in C$, nous ajoutons trois *copies de sommets* $b^c_{x,y}$, $b^c_{y,z}$ et $b^c_{z,x}$ toutes copies de b_G. Pour chaque littéral $a \in c$ construit à partir d'une variable u (c'est-à-dire tel que $a = u$ ou $a = \bar{u}$), nous ajoutons :
  - une *copie de sommet* v^c_a de v_u ;
  - si $a = u$, deux arcs $e^c_a = (v^c_a, b^c_{a,.})$ et $e'^c_a = (v^c_a, b^c_{.,a})$, de poids respectifs $(1, 0)$ et $(1, 1)$ (c'est-à-dire le même poids dans la première dimension que l'arc e_u) ;

– si $a = \bar{u}$, deux arcs $e_a^c = (b^c_{a,.}, v_a^c)$ et $e'^c_a = (b^c_{.,a}, v_a^c)$, de poids respectifs $(0,0)$ et $(0,1)$ (donc le même poids dans la première dimension que $e_{\bar{u}}$).

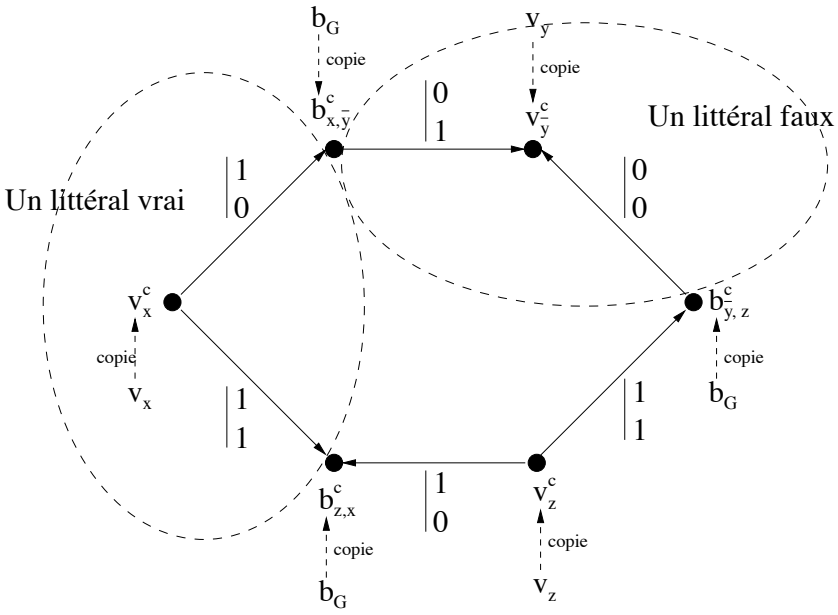


FIG. 4.15 – Transformation d'une clause $c = \{x, \bar{y}, z\}$.

Bien sûr, la transformation ci-dessus est polynomiale. Pour chaque variable, nous ajoutons une structure de *copie de sommet*, un sommet et deux arcs au graphe, et pour chaque clause, nous ajoutons six structures de *copie de sommet* et six arcs au graphe. Chaque copie représente un sommet et quatre arcs, ce qui nous fait un total de $2|U| + 6|C| + 1$ sommets et $6|U| + 30|C|$ arcs, soit un graphe de taille $O(|U| + |C|)$.

Réduction Soit (U, C) une instance de 3SAT et $G = (V, E, d) = f(U, C)$. Nous devons prouver que :

– si (U, C) est une instance positive de 3SAT, alors G est une instance positive de PARALLÉLISATION INTERNE 2D. Soit T un assignement de vérité pour U satisfaisant toutes les clauses de C. Nous posons :

$$r(b_G) = 0$$
$$\forall u \in U,\ r(b_u) = 0 \text{ et } r(v_u) = \begin{cases} 1 & \text{si } T(u) = F \\ 0 & \text{si } T(u) = V \end{cases}$$
$$\forall c \in C,\ r(b^c_{.,.}) = 0 \text{ et } \forall a \in c \text{ construit à partir de } u \in U,\ r(v_a^c) = r(v_u)$$

Nous montrons alors que r, considéré comme un *retiming* de la première dimension de G, est légal et que $\tilde{G}_r$ ne contient aucun cycle de poids non nul (en fait, il n'a pas de cycle du tout) :

– par définition de r, toutes les copies d'un sommet donné (utilisant la structure de *copie de sommet*) ont la même valeur de *retiming* que leur original, donc le poids des arcs de chaque structure de *copie de sommet* reste inchangé. Par conséquent, ces arcs restent portés par la boucle externe ; ils n'apparaissent pas dans $\tilde{G}_r$. Donc aucun des sommets construits par copie n'est connecté à son original dans $\tilde{G}_r$;
– pour chaque littéral $u \in U$, soit $r(b_u) = r(v_u) = 0$, $d_r(e_u) = (1, 0)$ et $d_r(e_{\bar{u}}) = (0, 0)$, soit $r(b_u) = 0$, $r(v_u) = 1$, $d_r(e_u) = (0, 0)$ et $d_r(e_{\bar{u}}) = (1, 0)$. Quel que soit le cas, exactement un

seul des deux arcs correspondant à un littéral appartient à $\tilde{G}_r$. De plus, les sommets b_u et v_u sont connectés au reste du graphe uniquement par des structures de *copie de sommet*. Ainsi, en utilisant la remarque précédente, nous en déduisons qu'ils ne sont connectés à rien d'autre dans $\tilde{G}_r$. L'arc restant (e_u ou $e_{\bar{u}}$) est donc isolé dans $\tilde{G}_r$; il ne peut faire partie d'aucun cycle. En outre, par définition de $r(v_u)$, remarquons que pour tout littéral a, $d_r(e_a) = (1, 0)$ si $T(a) = V$ et $d_r(e_a) = (0, 0)$ si $T(a) = F$;

– par construction, pour chaque clause $c \in C$ et chaque littéral $a \in c$ (avec soit $a = u$, soit $a = \bar{u}$, où $u \in U$), les deux arcs e_a^c et e'^c_a correspondants ont le même poids que e_a dans leur première dimension. De plus, par définition de r ($r(v_a^c) = r(v_u)$ et $r(b^c_{.,a}) = r(b^c_{a,.}) = r(b_u) = 0$) et par choix de la direction de ces arcs lors de leur construction, e_a^c et e'^c_a ont les mêmes valeurs de *retiming* que e_a pour leur source et leur puits. Puisqu'au moins un littéral est vrai par clause, au moins un arc – en réalité une paire d'arcs « jumeaux » construits pour un littéral – du cycle généré par l'ensemble de la clause c possède un poids strictement positif dans sa première dimension. Ainsi ce cycle, généré pour la clause, est coupé dans $\tilde{G}_r$ par au moins un arc. Finalement, puisque tous les sommets de la clause sont obtenus par l'utilisation de structures de *copie de sommet*, ce cycle n'est connecté à rien d'autre dans $\tilde{G}_r$.

En résumé, $\tilde{G}_r$ ne possède aucun cycle : c'est un ensemble de chaînes non dirigées composées de 0, 1, 2 ou 4 arcs (selon qu'il s'agisse d'arcs correspondant à une variable ou au cycle d'une clause et selon le nombre de ces arcs portés par la boucle externe). De plus, tous les poids après *retiming* sont positifs ou nuls, donc r est un *retiming* légal de G. Le graphe G est donc une instance positive de Parallélisation interne 2d.

– si G est une instance positive de Parallélisation interne 2d alors (U, C) est une instance positive de 3SAT. Soit r un *retiming* légal de G dans la première dimension seulement tel que G_r n'ait aucun cycle de poids non nul. Nous posons :

$$\forall u \in U, T(u) = \begin{cases} V & \text{si } d_r(e_u)_1 > 0 \\ F & \text{sinon} \end{cases} \quad \text{et } T(\bar{u}) = \begin{cases} V & \text{si } d_r(e_{\bar{u}})_1 > 0 \\ F & \text{sinon} \end{cases}$$

Nous commençons par donner quelques propriétés liées à ce *retiming* de G :

– pour chaque sommet y généré comme copie d'un sommet x (par une structure de *copie de sommet*), par légalité du *retiming* r, les valeurs de *retiming* de x et y diffèrent d'au plus 1, autrement une valeur lexico-négative apparaîtrait. De plus, si ces valeurs diffèrent de 1, alors une des deux paires d'arcs d'un des sommets vers l'autre devient de poids nul dans sa première dimension et cette paire apparaît donc dans $\tilde{G}_r$. Or, par construction, cette paire d'arcs forme un cycle de poids non nul dans sa seconde dimension, c'est-à-dire dans $\tilde{G}_r$. Ceci contredit l'hypothèse que r est une solution à notre instance, donc $r(x) = r(y)$: deux sommets séparés par une structure de *copie de sommets* ont la même valeur de *retiming*.
– pour toute clause $c \in C$ et pour tout littéral $a \in c$ construit à partir d'une variable $u \in U$ (soit $a = u$, soit $a = \bar{u}$), puisque les sommets $b^c_{.,.}$ et v_a^c sont respectivement des copies de b_G et v_u (par structure de *copie de sommets*), nous avons $r(b^c_{.,.}) = r(b_G) = r(b_u)$ et $r(v_a^c) = r(v_u)$. Dans G, les arcs e_a^c et e'^c_a, produits pour a ont le même poids dans leur première dimension que e_a (par construction). En outre, comme nous venons de le montrer, ils ont tous la même valeur de *retiming* pour leur source et leur puits. Ainsi, dans G_r, les deux arcs e_a^c et e'^c_a ont le même poids que e_a dans la première dimension.

Nous pouvons alors prouver que T est un assignement de vérité (c'est-à-dire $T(u) \neq T(\bar{u})$) et qu'il satisfait toutes les clauses de C :

- pour chaque variable $u \in U$, $d(e_u) = (1,0)$ et $d(e_{\bar{u}}) = (0,0)$. Puisque r est légal et puisque le poids d'un circuit ne change pas par *retiming*, nous avons soit $d_r(e_u) = (1,0)$ et $d_r(e_{\bar{u}}) = (0,0)$, soit $d_r(e_u) = (0,0)$ et $d_r(e_{\bar{u}}) = (1,0)$. Par définition de T, ceci prouve que $T(u) \neq T(\bar{u})$.
- $\tilde{G}_r$ n'a aucun cycle de poids non nul. Par construction, le cycle construit pour une clause a un poids non nul dans sa seconde dimension. Effectivement, dans cette seconde dimension, trois 1 se trouvent dans ce cycle, donc quelle que soit la direction prise pour les arcs le poids du cycle ne peut être que -3, -1, 1, ou 3. Ainsi, ce cycle doit forcément être coupé dans $\tilde{G}_r$: il existe au moins un arc avec un poids strictement positif dans sa première dimension. Soit e un tel arc : e correspond à la copie d'un littéral a, et comme nous l'avons montré précédemment, il a le même poids dans sa première dimension que e_a. Donc e_a a un poids strictement positif dans sa première dimension et par définition $T(a)$ est vrai. Ce raisonnement peut être tenu pour chaque clause, donc elles ont toutes au moins un littéral vrai et sont satisfaites.

En résumé, T est donc un assignement de vérité satisfaisant toutes les clauses de C : (C,U) est une instance positive de 3SAT.

Ceci termine la preuve : PARALLÉLISATION INTERNE 2D est NP-complet au sens fort. □

4.4.3 L'algorithme de Lang, Passos et Sha

Nous présentons dans cette section un contre-exemple qui prouve que l'algorithme présenté dans [69] est faux. Cet algorithme ne remet donc pas en cause notre preuve de NP-complétude. Nous commençons par rappeler les définitions formulées dans [69] et utiles à la compréhension du contre-exemple.

- dans la section *Basic concepts, problem model* de [69] :
  « Définition 1 : Un graphe de dépendance de boucle multi-dimensionnel (ou MLDG) $G = (V, E, \delta_L, D_L)$ est un graphe dirigé pondéré sur ses sommets et ses arcs, où V est l'ensemble des nids les plus internes, $E \subseteq V \times V$ est l'ensemble des arcs de dépendance entre les boucles, δ_L est une fonction de E dans $\mathbb{Z}^n$, représentant le vecteur de dépendance minimal, selon l'ordre lexicographique, associé à un arc, et D_L est une fonction de E dans $2^{\mathbb{Z}^n}$, représentant l'ensemble des vecteurs de dépendance entre deux sommets, et où n est le nombre de dimensions. »[4] ;
- dans la section *Basic concepts, problem model* de [69] :
  « Le cas particulier où deux vecteurs de dépendance ou plus entre deux sommets sont inférieurs à $(1, -\infty)$. [...] de tels arcs sont appelés arcs difficiles pour le parallélisme, ou simplement arcs difficiles. »[5].
  Cette définition correspond à un cycle (ou plusieurs cycles s'il y a plus de deux arcs) formé par deux arcs dont le poids est non nul dans la seconde dimension et nul dans la première. Notons que cette définition ne correspond pas à l'ensemble de tous les cycles de poids non nul dans la seconde dimension ;

[4] *Definition 1 A multi-dimensional loop dependence graph (MLDG) $G = (V, E, \delta_L, D_L)$ is a node-weighted and edge-weighted directed graph, where V is the set of innermost loop nests, $E \subseteq V \times V$ is the set of dependence edges between the loops, δ_L is a function from E to $\mathbb{Z}^n$,representing the minimal loop dependency vector in lexicographic order associated with an edge, and D_L is a function from E to $2^{\mathbb{Z}^n}$, representing the set of loop dependency vectors between two nodes, where n is the number of dimensions.*

[5] *The particular case when there are two or more different loop dependence vectors between two nodes which are less than $(1, -\infty)$. [...] such edges are called parallelism hard edges, hard edges for short.*

– dans la section *Parallel loop fusion, fully parallel loop fusion* de [69] :
« Définition 3 : étant donné un 2LDG $G = (V, E, \delta_L, D_L)$ et un vecteur h de dimension 2, $G - h = (V, E, w'')$ où $w''(e) = \delta_L(e) - h$ pour chaque arc difficile et $w''(e) = \delta_L(e)$ pour tous les autres arcs. »[6].

Nous pouvons alors retranscrire leur théorème qui donne une condition suffisante et nécessaire pour qu'un graphe soit parallélisable par *retiming*. Notre contre-exemple va montrer que ce théorème est faux.

– théorème faux de la section *Parallel loop fusion, fully parallel loop fusion* de [69] :
« Théorème 10 : Soit un 2LDG $G = (V, E, \delta_L, D_L)$. Il existe un *retiming* légal r tel que, après *retiming* et fusion de boucles, la boucle interne soit DO ALL si et seulement si chaque circuit de $G - (1, 0) = (V, E, w'')$ satisfait $w''(c) \geq (0, -\infty)$. »[7].

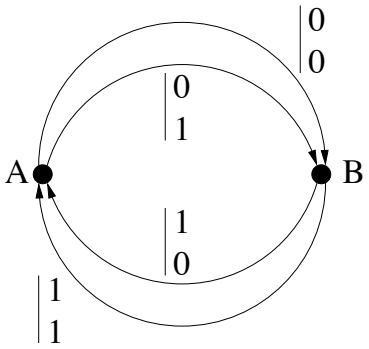


FIG. 4.16 – La condition n'est pas suffisante.

Le graphe de la figure 4.16, est un graphe satisfaisant la condition énoncée par le théorème 10 de [69] mais qui n'est pas parallélisable. En effet, quel que soit le *retiming* dans la première dimension, une des deux paires d'arcs entre les deux sommets restera toujours non portée par la boucle externe. Comme ces deux paires forment chacune un cycle de poids non nul dans la seconde dimension, le graphe n'est pas parallélisable. Dans ce graphe, les arcs difficiles sont les arcs de A vers B, donc le graphe $G - (1, 0)$ est composé de deux arcs de A vers B de poids respectifs $(-1, 0)$ et $(-1, 1)$ et de deux arcs de B vers A de poids respectifs $(1, 0)$ et $(1, 1)$. Le circuit de poids minimal dans $G - (1, 0)$ a un poids de $(0, 0)$, il satisfait donc la condition du théorème 10 de [69]. Ceci prouve donc que ce théorème est faux. L'erreur vient du fait que tous les cycles – ceux qui ne sont pas portés aussi bien que ceux qui sont portés par la boucle externe – doivent être considérés pour la parallélisation.

La condition est donc nécessaire (nous pouvons l'admettre) mais pas suffisante. L'algorithme que Lang, Passos et Sha déduisent de ce théorème est donc faux. En bref, le principe de cet algorithme est de trouver un *retiming* légal $r[1]$ de la première dimension de $G - (1, 0)$ (en utilisant l'algorithme 2), puis de compléter ce *retiming* par un *retiming* $r[2]$ de la seconde dimension rendant nulles toutes les dépendances de $\tilde{G}_{r[1]}$. Le deuxième *retiming* $r[2]$ ne pourra pas être trouvé lorsque $r[1]$ ne coupe pas tous les cycles de $\tilde{G}_{r[1]}$. Or, à cause de la construction de $G - (1, 0)$ dans [69], rien ne garantit que $r[1]$ va effectivement couper tous les cycles en question. Dans le cas de notre exemple en

[6] *Definition 3 Given a 2LDG $G = (V, E, \delta_L, D_L)$ and a two dimensional vector h, $G - h = (V, E, w'')$ where $w''(e) = \delta_L(e) - h$ for each hard edge and $w''(e) = \delta_L(e)$ for any other edge.*

[7] *Theorem 10 Given a 2LDG $G = (V, E, \delta_L, D_L)$, then there is a legal retiming r such that, after retiming and loop fusion, the innermost loop is DO ALL iff each cycle in $G - (1, 0) = (V, E, w'')$ satisfies $w''(c) \geq (0, -\infty)$.* Notons que nous avons traduit le terme *cycle* de [69] par circuit. La lecture de l'article confirme que Lang, Passos et Sha l'emploient dans ce sens.

figure 4.16, le *retiming* de la dimension externe va découvrir un cycle de poids non nul qui empêche la parallélisation subséquente. Autrement dit, l'algorithme peut trouver une solution fausse.

La solution dans ce cas n'est pas d'étendre la définition des arcs difficiles aux arcs portés par la boucle externe. De façon générale, la seule manière de résoudre le problème est bel et bien de considérer tous les cycles du graphe. C'est cette nécessité qui rend le problème NP-complet.

4.4.4 Formulation par contraintes linéaires

Dans cette section, nous proposons un ensemble de contraintes linéaires dont la résolution en nombres entiers nous fournit une solution exacte au problème de parallélisation interne. Bien entendu, cette résolution est en théorie exponentielle, cependant le faible nombre de contraintes la rend envisageable dans des cas pratiques sur de petites instances du problème. Pour des raisons de simplicité, nous limitons le problème au cas d'une simple boucle séquentielle entourant un nid totalement parallèle. Grâce au théorème 5, ce cas regroupe tous les cas de recherche d'un ensemble de boucles séquentielles entourant un ensemble de boucles totalement parallèles.

Avant de donner notre formulation par contraintes linéaires, nous montrons un résultat intermédiaire qui nous permettra de prouver que notre système résout bien toutes les instances positives. Ce résultat est une version affaiblie du lemme 7 mais qui est valable pour tout *retiming* (légal ou non).

Lemme 8 *Soit un graphe de dépendance $G = (V, E, d)$, pour tout* retiming *r de G il existe un* retiming *r' de G tel que $d_r(e) = 0 \Rightarrow d_{r'}(e) = 0$ (tous les arcs de poids nul dans G_r ont également un poids nul dans $G_{r'}$) et tel que $\forall v \in V$, $0 \leq r'(v) \leq \sum_{\{e \in E | d_r(e)=0\}} |d(e)|$.*

Preuve Nous construisons un sous-graphe $G' = (V, E', d)$ de G à partir de r en ne conservant que les arcs dont le poids est nul dans G_r. Par construction, tous les arcs de G'_r ont un poids nul. Pour chaque composante connexe s de G', nous choisissons un sommet u_s ayant la valeur de *retiming* $r(u_s)$ minimale dans s. Nous définissons alors pour chaque sommet $v \in s$, $r'(v) = r(v) - r(u_s)$. Pour tout arc $e = (u, v)$ de G', u et v appartiennent à la même composante connexe s, donc $r(u)$ et $r(v)$ ont varié de la même quantité (nous leur avons enlevé $r(u_s)$). Par conséquent, le poids de e est le même pour les deux *retimings* : $d_r(e) = d_{r'}(e) = 0$. De plus, pour chaque composante connexe s de G', le sommet u_s a pour valeur de *retiming* $r'(u_s) = 0$ et, par choix de u_s, tous les autres sommets $v \in s$ sont tels que $r'(v) \geq 0$. Enfin, pour tout arc $e = (u, v)$ de s, $r'(u) = r'(v) + d(e)$ (poids nul après *retiming*). Donc, pour tout sommet $v \in s$, $r'(v)$ est le poids d'un chemin élémentaire non dirigé de v à u_s dans G'. Donc $0 \leq r'(v) \leq \sum_{e \in E'} |d(e)|$. □

Nous sommes alors en mesure de proposer notre système d'inéquations linéaires et de prouver que sa résolution est équivalente à la résolution de notre problème.

Proposition 15 *Soit un graphe de dépendance $G = (V, E, d)$ de dimension n et $M_x = \sum_{e \in E} |d(e)_x|$ pour tout $2 \leq x \leq n$. Il existe un* retiming *multi-dimensionnel légal r tel que les $(n-1)$ boucles les plus internes soient parallèles dans G_r si et seulement si le système d'inéquations suivant admet une solution entière :*

$$
\begin{array}{rl}
\forall e = (u, v) \in E, & M_2(d(e)_1 + r_1(v) - r_1(u)) + (d(e)_2 + r_2(v) - r_2(u)) \geq 0 \\
& M_2(d(e)_1 + r_1(v) - r_1(u)) - (d(e)_2 + r_2(v) - r_2(u)) \geq 0 \\
& \vdots \\
& M_n(d(e)_1 + r_1(v) - r_1(u)) + (d(e)_n + r_n(v) - r_n(u)) \geq 0 \\
& M_n(d(e)_1 + r_1(v) - r_1(u)) - (d(e)_n + r_n(v) - r_n(u)) \geq 0
\end{array}
$$

De plus, toute solution entière r à ce système est un retiming *multi-dimensionnel légal de G tel que les $(n-1)$ boucles les plus internes sont parallèles dans G_r.*

Preuve Remarquons tout d'abord que, dans ce système d'inéquations, chaque groupe de deux inégalités est équivalent à l'inégalité suivante :

$$M_x d_r(e)_1 \geq |d_r(e)_x|$$

Cette dernière forme est certainement plus simple à comprendre intuitivement : si le poids dans la première dimension est nul, alors la valeur absolue dans la $x^{ème}$ devra être nulle, sinon elle pourra prendre « n'importe quelle valeur ».

Supposons d'abord que le système ait une solution r. Pour tout arc $e = (u, v)$, les deux premières contraintes impliquent que $2M_2(d(e)_1 + r_1(v) - r_1(u)) \geq 0$. Nous pouvons supposer que $M_2 > 0$ (autrement tous les poids sont nuls dans la deuxième dimension et la deuxième boucle est parallèle). Donc $d(e)_1 + r_1(v) - r_1(u) \geq 0$, et r est un *retiming* légal de la première dimension. Deux cas se présentent alors : soit $d(e)_1 + r_1(v) - r_1(u) = 0$ et dans ce cas les contraintes mènent à $d(e)_x + r_x(v) - r_x(u) = 0$ pour tout $x \geq 2$ et la dépendance est indépendante de la boucle après un *retiming* par r, soit $d(e)_1 + r_1(v) - r_1(u) > 0$ et la dépendance est portée par la première boucle tandis que $d(e)_x + r_x(v) - r_x(u)$ peut avoir n'importe quelle valeur (sans incidence sur le parallélisme) pour tout $x \geq 2$. Donc r est un *retiming* légal de G tel que les $(n-1)$ boucles les plus internes de G_r sont parallèles.

Inversement, supposons qu'il existe un *retiming* légal r de G tel que les $(n-1)$ boucles les plus internes de G_r soient parallèles. Puisque r est légal, pour tout arc $e = (u, v)$, $d(e)_1 + r_1(v) - r_1(u) \geq 0$. De plus, $\tilde{G}_r$ ne contient que des arcs de poids nul. Mais, si r est une solution quelconque, il est possible que, pour les boucles internes, les poids après *retiming* de certains arcs (ceux ne se trouvant pas dans $\tilde{G}_r$) soient trop grands pour satisfaire les contraintes du système d'inéquations. Nous montrons alors qu'il est possible de déduire de toute solution une autre solution vérifiant les contraintes.

Soit G' le sous-graphe de G formé par les arcs contenus dans $\tilde{G}_r$: par définition de r, G'_r ne contient que des arcs de poids nul. Pour chaque dimension $x \geq 2$, le lemme 8 nous permet de trouver un *retiming* r'_x de la $x^{\text{ième}}$ dimension de G' tel que pour tout arc e de G', $d_r(e)_x = 0 \Rightarrow d_{r'}(e)_x = 0$ et $\forall v \in V, 0 \leq r'_x(v) \leq \sum_{\{e \in G' | d_r(e)_x = 0\}} |d(e)_x|$. Nous définissons un nouveau *retiming* multi-dimensionnel R de G tel que $R_1(v) = r_1(v)$ et $R_x(v) = r'_x(v)$. Si $d(e)_1 + r_1(v) - r_1(u) = 0$, e est dans $\tilde{G}_r$ et pour tout $x \geq 2$, $d(e)_x + r_x(v) - r_x(u) = 0$ (car r est solution) donc par construction de r'_x, $d(e)_x + r'_x(v) - r'_x(u) = 0$. Si $d(e)_1 + r_1(v) - r_1(u) > 0$, e n'est pas dans $\tilde{G}_r$ et pour tout $x \geq 2$, nous avons d'une part $d(e)_x + r'_x(v) - r'_x(u) \leq d(e)_x + \sum_{e \in \tilde{G}_r} |d(e)_x| \leq M_x$ car $e \notin \tilde{G}_r$, et d'autre part $d(e)_x + r'_x(v) - r'_x(u) \geq d(e)_x - \sum_{e \in \tilde{G}_r} |d(e)_x| \geq -M_x$, là encore car $e \notin \tilde{G}_r$. Donc, toutes les contraintes sont satisfaites et le système admet R pour solution. □

Ainsi, grâce à ce système d'inéquations linéaires, nous sommes capables de trouver une solution comportant une boucle séquentielle entourant un nid complètement parallèle. Le système reste relativement petit avec $n|V|$ variables (le *retiming*) et $2n|E|$ contraintes. Notons qu'il est tout à fait possible d'étendre cette méthode afin qu'une boucle à l'intérieur de ce nid reste séquentielle. Il suffit d'enlever la contrainte comportant une soustraction pour cette boucle. En procédant ainsi, le vecteur de distance conserve une contrainte de lexico-positivité, mais une valeur nulle n'est plus obligatoire lorsque la dépendance n'est pas portée par la boucle externe. Cependant, avec cette méthode, la seconde boucle séquentielle ne permet pas d'aider à paralléliser les boucles plus internes encore. D'autre extensions sont possibles en gardant en tête l'idée principale : la constante M_x dans le programme sert à donner plus d'importance au poids dans la première dimension. D'une certaine manière, elle simule l'ordre lexicographique sous forme de contraintes linéaires.

4.5 Cas polynomiaux et heuristiques

L'approche exacte proposée en section 4.4.4 ayant une complexité exponentielle, nous présentons ici un certain nombre d'algorithmes polynomiaux cherchant à identifier certaines configurations solubles du problème. Afin de rendre la discussion plus simple, nous nous restreignons toujours au cas de deux boucles pour lesquelles nous souhaitons trouver un *retiming* rendant la boucle interne parallèle. Nos deux premiers algorithmes identifient des cas pour lesquels le problème de parallélisation est soluble en temps polynomial. Le dernier algorithme est une méthode constructive pour la résolution du cas général (qui peut permettre, par exemple, de mettre en œuvre une recherche non exhaustive d'une solution). Enfin, nous donnons également dans cette section une condition suffisante à la non satisfaisabilité d'une instance du problème.

4.5.1 Boucle parallèle interne par *retiming* interne seul

Il s'agit du cas le plus simple, dans lequel toutes les dépendances de $\tilde{G}$ peuvent être transformées directement en dépendances indépendantes de la boucle. Autrement dit, il s'agit du cas où la boucle interne est parallélisable par l'algorithme 11, déjà étudié en section 4.3. La figure 4.17 illustre cette résolution ($\tilde{G}$ est le graphe des arcs tracés en traits pleins). Notons que sur cet exemple la technique du *retiming* externe seul (voir section suivante) est inefficace. L'exemple de Peir et Cytron (exemple 1) est un autre cas de figure où le *retiming* interne seul suffit comme illustré sur les figures 4.1 et 4.3.

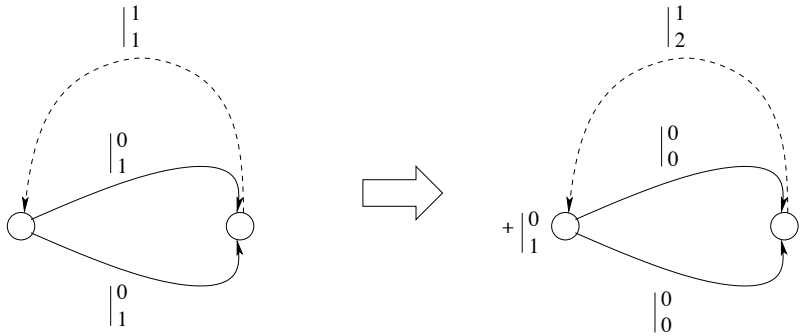


FIG. 4.17 – Parallélisation par *retiming* interne seul.

4.5.2 Boucle parallèle interne par *retiming* externe seul

Si toutes les dépendances de $\tilde{G}_r$ ont un poids nul, alors la boucle interne est parallèle. Plutôt que de chercher un *retiming* externe permettant de trouver un $\tilde{G}_r$ sans circuits de poids non nul (ce qui est NP-complet), nous pouvons chercher un *retiming* externe permettant de trouver un $\tilde{G}_r$ dont tous les arcs ont déjà un poids nul (sans *retiming* interne additionnel). Autrement dit, nous souhaitons trouver un *retiming* r de G tel que pour tout arc $e = (u, v)$ de G, $d(e)_1 + r_1(v) - r_1(u) \geq 1$ si $d_2(e) \neq 0$ et $d(e)_1 + r_1(v) - r_1(u) \geq 0$ sinon. Ceci peut être résolu en temps polynomial par l'algorithme 2 de *retiming* sous contraintes se trouvant à la page 22. La figure 4.18 est un exemple de graphe pour lequel cet algorithme peut être utilisé avec succès. Remarquons que cet exemple ne peut pas être résolu par *retiming* interne seul (voir section précédente).

Cette technique échoue lorsque le poids des circuits dans la première dimension est trop faible : trop peu d'arcs peuvent alors être portés par la boucle externe pour éliminer tous les arcs ayant

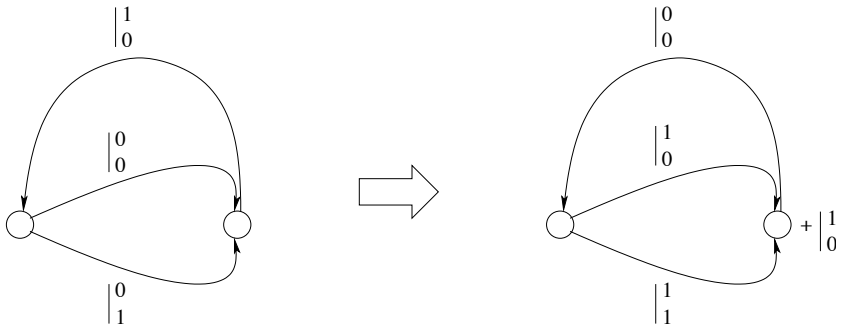


FIG. 4.18 – Parallélisation par *retiming* externe seul.

un poids non nul dans la seconde dimension. C'est le cas de l'exemple de Peir et Cytron (voir la figure 4.1). En revanche, cette technique fonctionne toujours pour un graphe sans circuits, ce qui est le cas, entre autres, pour de nombreuses applications de traitement du signal.

En reconsidérant la question du *retiming* externe seul, nous venons de classifier les questions suivantes : existe-t-il un *retiming* r de la première dimension de G tel que $\tilde{G}_r$:

1 n'ait aucun arc ?
2 n'ait que des arcs de poids nul ?
3 soit un ensemble de chaînes (non dirigées) ?
4 soit une forêt ?
5 n'ait aucun cycle de poids non nul ?

Notre preuve de NP-complétude présentée en section 4.4.2 prouve que les points 3, 4 et 5 sont des problèmes NP-complets, tandis les points 1 et 2 sont polynomiaux. La question 2 est donc, parmi les problèmes que nous avons considérés, le problème de *retiming* externe pour la parallélisation le plus général qui soit soluble en temps polynomial.

4.5.3 Une heuristique multi-dimensionnelle

Lorsque le *retiming* dans une seule dimension est insuffisant pour trouver une boucle parallèle, il reste à tenter une véritable approche multi-dimensionnelle. L'idée est de renverser le problème en commençant par déterminer le $\tilde{G}_r$ que nous souhaitons obtenir, puis à chercher le r correspondant, s'il existe. Nous savons quels graphes $\tilde{G}_r$ nous conviennent : ce sont les graphes pour lesquels il existe un *retiming* rendant le poids de tous leurs arcs nul. Ces graphes sont caractérisés tout d'abord par le lemme 6, qui prouve que les graphes sans cycles (comme les arbres couvrants par exemple) permettent toujours l'existence d'un tel *retiming*, puis par le théorème 4 qui montre la même chose pour les graphes sans cycles de poids non nul. Pour cette heuristique, la procédure est donc la suivante :

- choisir $T(G)$ un arbre couvrant de G ;
- transformer $T(G)$ en G' en y ajoutant tous les arcs de G ne formant pas de cycle de poids non nul avec $T(G)$ dans la seconde dimension ;
- à l'aide de l'algorithme 2, chercher un *retiming* r de la première dimension tel que pour tout arc $e \in E$, $d_r(e) \geq 0$ si e est un arc de G' et $d_r(e) \geq 1$ sinon ;
- compléter r par un alignement de la seconde dimension.

La figure 4.19 illustre cette procédure. Comme nous pouvons le constater, le choix de l'arbre couvrant est primordial puisque sur la figure en question, deux choix différents mènent l'un à une solution et pas l'autre. Notons que, sur cet exemple, un *retiming* dans une seule dimension seulement (aussi

bien externe qu'interne) est insuffisant.

Remarquons enfin que toute solution au problème général correspond à un $\tilde{G}_r$ particulier, donc à un arbre couvrant de ce $\tilde{G}_r$ complété d'un certain nombre d'arcs. Cette heuristique nous fournit

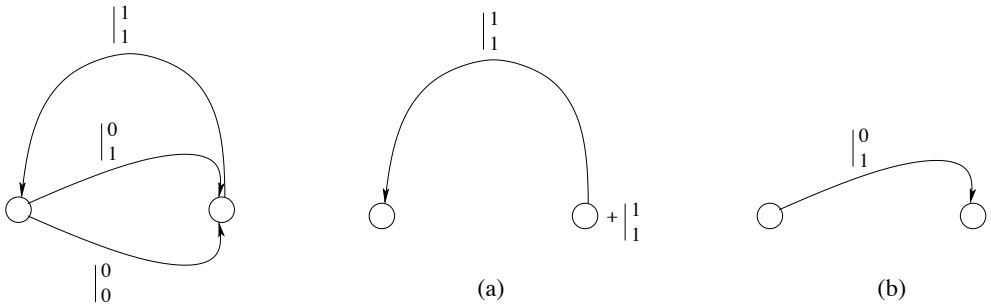


FIG. 4.19 – Deux différents arbres couvrants : a) mène à une solution b) ne mène pas à une solution.

donc également un second moyen de résolution exacte du problème : par énumération de tous les arbres couvrants de G. Évidemment, là encore, cette solution exacte est de complexité exponentielle (puisque le nombre d'arbres couvrants est lui-même exponentiel).

4.5.4 Cas insoluble

Nous terminons cette section par un cas particulier de graphe pour lequel nous pouvons être sûrs qu'il n'existe pas de *retiming* rendant la boucle interne parallèle. C'est le cas de graphes possédant un circuit c de poids $0 <_{lex} w(c) \leq_{lex} (0, +\infty)$. En effet, quel que soit le *retiming* légal choisi pour la première dimension, tous les arcs de c appartiennent à $\tilde{G}_r$ (à cause de la conservation par *retiming* du poids du circuit et de la légalité). Donc, dans tous les cas, $\tilde{G}_r$ a un circuit (donc un cycle) de poids non nul et la parallélisation est impossible. Notons bien sûr que cette condition est suffisante mais non nécessaire pour garantir qu'un graphe n'est pas parallélisable.

Un tel cas peut assez facilement être détecté. Il faut commencer par s'assurer de la lexico-positivité de tous les arcs en choisissant un *retiming* légal de G. Dans ce cas, si un circuit a un poids nul dans sa première dimension, puisque le *retiming* est légal, alors tous les arcs de ce circuit auront la première composante de leur poids égale à zéro. Donc, quel que soit le *retiming* légal r choisi, tous les arcs d'un tel circuit se retrouvent alors dans $\tilde{G}_r$. Nous n'avons plus qu'a vérifier si $\tilde{G}_r$ possède un circuit ou non. La figure 4.20 illustre un tel cas insoluble.

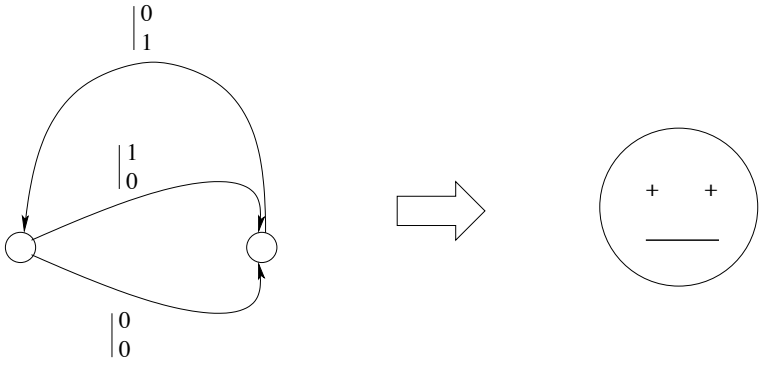


FIG. 4.20 – Un exemple non parallélisable.

4.6 Conclusions et extensions possibles

Le but initial de ce chapitre était d'étudier sous quelles conditions il est possible de paralléliser un certain nombre de boucles à l'intérieur d'un nid en ayant uniquement recours à des transformations simples comme la fusion/distribution ou le décalage d'instructions. Ces techniques ont un comportement commun vis-à-vis du domaine d'itération du nid de boucle : elle ne changent pas la structure de celui-ci (entre autres son ordre de parcours), mais effectuent des translations sur les instructions qu'il contient. Alors que les techniques de fusion/distribution ont été intensément étudiées, peu de choses étaient connues sur le décalage d'instructions pour la parallélisation de boucles. C'est pourquoi nous souhaitions mieux comprendre ses mécanismes, afin d'être à même de produire des solutions à base de décalage seul ou de combinaisons plus élaborées (décalage/torsion par exemple). Disposer d'une telle technique était également motivé par le fait qu'un algorithme plus puissant et plus général produit souvent une transformation plus complexe que nécessaire et la plupart du temps n'inclut pas comme solutions possibles celles qu'un décalage seul peut produire.

4.6.1 Résumé des résultats

Une partie du problème général, que nous avons renommée décalage interne, est déjà bien connue : il s'agit du problème de l'alignement étudié par Peir [89]. Nous l'avons rappelé ici car il constitue l'élément de base de la généralisation au cas multi-dimensionnel. Le résultat principal de ce chapitre concerne donc le cas multi-dimensionnel : la parallélisation de boucle utilisant un décalage dans toutes les dimensions d'un nid de boucles, afin de produire un certain nombre de boucles parallèles à l'intérieur de celui-ci. Nous avons montré que ce problème est NP-complet au sens fort. Ainsi, la simplicité du cas mono-dimensionnel disparaît lors de la combinaison du décalage dans plusieurs dimensions.

Ce résultat peut sembler à première vue surprenant puisque la plupart des algorithmes de détection de boucles parallèles (entre autres [68, 4, 108, 41, 42, 27]) sont polynomiaux (fondés sur la programmation linéaire en nombre rationnels) et beaucoup d'entre eux intègrent le décalage d'instructions. Si nous replaçons la parallélisation par décalage d'instructions dans la perspective de ces algorithmes de parallélisation classiques, nous pouvons mieux comprendre leurs différences. Les algorithmes classiques ont une stratégie extrêmement conservative : ils cherchent un vecteur d'ordonnancement permettant à toutes les dépendances d'être portées par la boucle externe. Ceci revient à imposer, pour tout circuit c du graphe, la contrainte : $x.d(c) \geq l(c)$ où x est le vecteur d'ordonnancement et $l(c)$ le nombre d'arcs de c. En termes de décalage, ces algorithmes s'arrangent donc pour toujours se trouver dans le cas 1 présenté dans la section 4.5.2 sur l'utilisation du *retiming* externe uniquement. Autrement dit, la partie décalage qu'ils intègrent n'est pas multi-dimensionnelle, et ils se privent d'une part des solutions rentrant dans le cas 2, et d'autre part des solutions multi-dimensionnelles mettant en jeu un décalage dans les deux dimensions. Évidemment, le problème vient de la contrainte trop forte qu'ils imposent à leur vecteur d'ordonnancement. L'idée serait donc d'imposer une contrainte plus faible, par exemple $x.d(c) \geq 1$ comme cela est proposé dans [26]. Cependant, dans ce dernier cas la contrainte est nécessaire mais en général insuffisante : il n'existe pas toujours de décalage complétant la transformation afin de produire une boucle interne parallèle. Nos résultats indiquent que trouver le bon compromis – quelque part entre les cas 2 et 3 – est NP-complet.

Malgré la complexité du problème, nous avons toutefois montré qu'une solution satisfaisait un système d'inéquations de taille modeste dont la résolution exacte peut être envisagée pour de petits problèmes. Nous avons également montré que le problème pouvait être partiellement résolu par plusieurs heuristiques polynomiales toutes fondées sur des parcours de graphes ou des variantes de

l'algorithme de Bellman-Ford.

Dans l'ensemble de ces résultats, nous avons supposé un cas de figure très simple : un nid de boucles à dépendances uniformes sur lequel nous n'appliquons que des décalages. Nous présentons maintenant un certain nombre d'extensions possibles qui s'intègrent très simplement au problème une fois le cas de base bien compris. Certaines de ces extensions s'inspirent fortement d'autres travaux en parallélisation de boucles, d'autres cherchent à généraliser les cas de figure que nous avons envisagés.

4.6.2 Augmenter la quantité de parallélisme à l'aide de duplication de code

Comme le montre Okuda dans [83], une duplication bien choisie du calcul combinée à un décalage peut permettre dans certaines situations de paralléliser une boucle pour laquelle un simple décalage est insuffisant. Okuda ne présente la technique que sur des structures de graphes particulières, mais l'idée sous-jacente est de casser les cycles de dépendances de $\tilde{G}_r$ en dupliquant les sources de ces cycles. Il est alors assez simple de réaliser que la duplication ne peut aider que lorsque $\tilde{G}_r$ est sans circuit (car un circuit est un cycle sans sources, il ne peut donc pas être cassé de cette manière).

Plus précisément, le principe de la duplication est de dupliquer un sommet du graphe (et donc le calcul correspondant) afin de pouvoir décaler de façon indépendante les deux copies du calcul. La technique originale d'Okuda n'utilise la duplication que sur les sommets qu'il qualifie d'« extrêmes », c'est-à-dire les sommets connectés à un unique autre sommet du graphe. Mais ce choix de sommet à répliquer n'est ni nécessaire ni suffisant. Comme nous avons pu le voir, le seul obstacle à la parallélisation est la présence de cycles de poids non nul dans $\tilde{G}_r$. Si le résultat d'un calcul est utilisé par deux instructions à venir, il suffit de le répliquer afin que chacune de ces deux instructions ne dépende que de sa copie indépendante. Ainsi, en dupliquant les sources de $\tilde{G}_r$ (et en remontant dans l'ordre topologique) il est possible de briser tous les cycles du graphe excepté les circuits. Nous pouvons donc utiliser un simple algorithme de parallélisation interne (donc applicable à $\tilde{G}$) fondé sur ce principe :

Algorithme 12 (*parallélisation par duplication + décalage*)

- partir d'un arbre couvrant $G' = T(G)$ de G ;
- pour chaque arc $e \in E$ n'appartenant pas à G', deux cas peuvent se produire :
  - e appartient à un cycle de poids nul : il peut être ajouté à G' sans modification ;
  - e appartient à un cycle de poids non nul : répliquer sa source et tous ses ancêtres dans G' permet de casser le cycle et e peut être ajouté à G'.

Le problème avec une telle méthode est qu'il est impossible de prédire la quantité de duplication dans le code final, mais le choix de l'arbre couvrant influence fortement cette quantité.

4.6.3 Combinaison de décalage et de transformations linéaires

Comme nous l'avons mentionné précédemment, notre technique de décalage peut aisément être combinée à une transformation linéaire de la boucle. En particulier, cela a été proposé dans [26] où le vecteur d'ordonnancement est choisi avec une contrainte plus faible et doit être complété par un *retiming*. Dans cette situation, le vecteur d'ordonnancement est toujours déterminé par programmation linéaire, donc imprévisible, mais l'ensemble des transformations envisagées par la procédure est plus grand. Cependant, comme nous l'avons déjà mentionné, la contrainte utilisée n'est pas toujours suffisante et il est possible que la partie linéaire ne puisse pas être complétée par un décalage rendant la boucle interne parallèle.

Une autre possibilité est d'appliquer une transformation linéaire fixe avant d'utiliser notre technique de décalage pour essayer de la compléter. Généralement, le nombre de transformations linéaires simples et intéressantes est relativement réduit (identité, permutations ou torsion avec un petit facteur). En effet, en pratique, la détection du parallélisme n'est pas le seul pré-requis à la réalisation d'un code efficace. D'autres facteurs, comme la distribution des calculs et des données, influencent la performance. Les algorithmes classiques détectent bel et bien le maximum de parallélisme, mais une transformation complexe ne se prête pas nécessairement aux autres optimisations dans la chaîne de compilation. C'est pourquoi il peut être intéressant de ne considérer que certaines transformations simples, afin d'être certains de pouvoir poursuivre les transformations ultérieures dans de bonnes conditions. Dans [63] nous pouvons trouver une méthode fondée sur ces idées : le principe est d'essayer d'appliquer plusieurs transformations linéaires candidates et d'évaluer les performances de chacune afin de choisir la meilleure. Dans ce contexte, notre technique de décalage s'intégrerait naturellement comme complément à la transformation linéaire.

Comme l'illustre la figure 4.21, le premier graphe ne peut pas être parallélisé par *retiming*[8], mais si nous utilisons un échange de boucles préalable le graphe devient alors parallélisable aussi bien par *retiming* interne qu'externe.

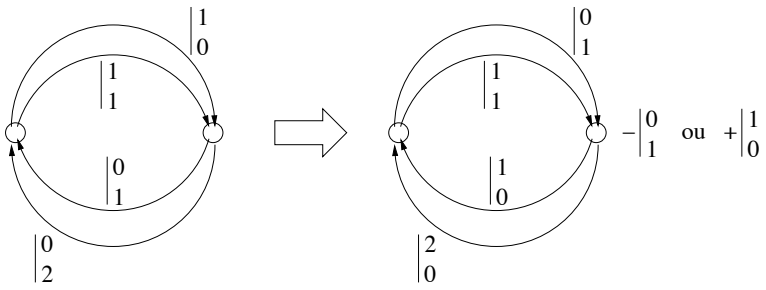


FIG. 4.21 – Cet exemple ne devient parallélisable qu'après une permutation de boucles.

4.6.4 Extension aux dépendances non uniformes

Dans le cas de dépendances non uniformes, nous pouvons étendre de façon limitée notre technique de *retiming*. Le *retiming* agissant comme une translation, nous pouvons agir sur un grand nombre de représentations des dépendances différentes ; cependant lorsqu'une dépendance représente un ensemble de plusieurs vecteurs de distance, si nous voulons garantir la parallélisme, nous devons trouver une (unique) translation telle que tous les vecteurs de cet ensemble deviennent soit nuls soit supérieurs à $(1, -\infty)$. Nous présentons ici la façon dont nous pourrions étendre notre technique aux vecteurs de direction (voir [109] pour une présentation détaillée des vecteurs de direction), ceci devrait donner un modèle suffisamment clair pour étendre le *retiming* aux autres représentations.

Les vecteurs de direction sont des vecteurs dont les composantes peuvent avoir pour valeur $*$ ou alors toute valeur entière éventuellement associée à un $+$ ou un $-$. La sémantique de ces différentes valeurs est la suivante :

- n correspond à la valeur entière n ;
- $n+$ correspond aux valeurs entières supérieures ou égales à n ($+$ désigne $1+$) ;
- $n-$ correspond aux valeurs entières inférieures ou égales à n ($-$ désigne $(-1)-$) ;
- $*$ correspond à toute valeur entière.

[8] Notons d'ailleurs que cet exemple ne répond pas à la condition énoncée en section 4.5.4, ceci illustre bien que cette condition n'est pas nécessaire.

Comme nous l'avons précisé précédemment, le *retiming* est une translation et ne peut donc agir que sur les parties entières de ces valeurs. Il en résulte que les valeurs pouvant devenir strictement nulles après *retiming* sont les valeurs possédant uniquement une partie entière (pas de $+$, $-$ ni de $*$), et les valeurs pouvant devenir strictement positives après *retiming* sont les valeurs ayant une composante entière et éventuellement un $+$ (pas de $-$ ni de $*$). Nous supposerons comme précédemment que tous les circuits ont un poids lexico-positif (défini en utilisant la somme habituelle sur les vecteurs de direction, c'est-à-dire par exemple $+$ plus $-$ donne $*$, $2+$ moins 3 donne $(-1)+$, etc.).

Le problème avec les vecteurs de direction est que l'algorithme 2 peut ne plus fonctionner : nous ne sommes plus systématiquement capables de produire un *retiming* légal à partir d'un graphe correct (sans circuit de poids strictement lexico-négatif). En effet, à cause des $-$ et des $*$, il est possible que certains arcs ne puissent pas avoir un poids positif ou nul à l'aide d'un *retiming* borné (évidemment, il est toujours possible de décaler sur la taille du domaine, mais dans ce cas nous faisons de la distribution). Une possibilité est alors de porter ces dépendances par une boucle externe. Ceci peut être fait de la même manière que nous l'avons fait dans le cas du *retiming* externe seul (section 4.5.2) ou du *retiming* sous contraintes (algorithme 2) : il suffit juste d'ajouter des contraintes plus fortes sur les arcs à porter (par exemple pour le cas en dimension 2, nous pouvons imposer au poids dans la première dimension d'être strictement positif lorsque la seconde dimension comporte un $-$ ou un $*$). Cependant, dans ce cas les contraintes sont plus fortes et il n'existe pas toujours de *retiming* les satisfaisant. Lorsque le *retiming* seul est incapable d'éliminer toutes les dépendances lexico-négatives et de produire un code complètement parallèle, il ne reste que le recours à la distribution, ce qui sort du cadre de cette thèse. La figure 4.22 donne deux exemples, le premier peut être parallélisé complètement par *retiming* seulement, le second non.

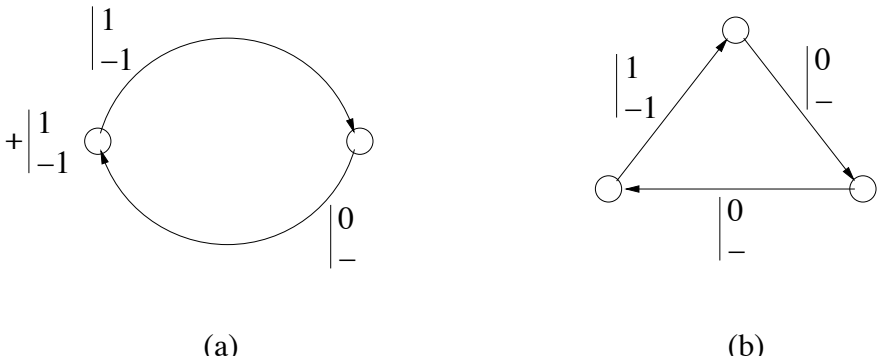


FIG. 4.22 – a) Fusion totale avec boucle interne parallèle possible b) impossible.

Chapitre 5

Localité

5.1 Introduction

Au chapitre 2, nous avons proposé un algorithme de minimisation du nombre d'arcs de poids nul par *retiming*. Dans ce chapitre, nous nous posons la question inverse de maximisation du nombre d'arcs de poids nul par *retiming*. Évidemment, cette question présente tout d'abord un intérêt intellectuel : alors que minimiser le nombre d'arcs de poids nul est un problème « facile », c'est-à-dire soluble en temps polynomial, il est intéressant de savoir si cela reste vrai pour le problème inverse. En outre, ce problème n'est pas complètement nouveau. Dans le contexte de l'alignement des données (ou distribution à la HPF), le placement des données est généralement exprimé par une fonction affine des indices d'accès aux éléments d'un tableau (voir [43, 30]). Le décalage correspond alors à la partie constante de cette fonction de placement. Le cas le plus simple, c'est-à-dire en dépendances uniformes et en prenant l'identité comme partie linéaire de la fonction de placement, est étudié dans [23]. Ce cas correspond à notre problème de maximisation pour un *retiming* quelconque (non nécessairement légal).

Dans le contexte de la transformation de programmes, un arc de poids nul correspond à un accès à une donnée calculée dans la même itération de la boucle. Maximiser le nombre d'arcs de poids nul va donc dans le sens de la réutilisation maximale des données. Bien que le problème ne corresponde pas directement à une fonction spécifique à une optimisation particulière, il représente une version simplifiée des problèmes communs à la maximisation de la localité et à la minimisation de la consommation mémoire. En outre, nous avons déjà rencontré deux tranformations allant dans le sens de la maximisation du nombre d'arcs de poids nul : la parallélisation de boucles, pour laquelle maximiser le nombre d'arcs de poids nul de la boucle interne peut aider à la paralléliser ensuite par *retiming* externe (voir section 4.5.2) et la fusion de boucles parallèles, pour laquelle un arc de poids non nul empêche la fusion (voir [22]). Ainsi, comprendre notre problème de maximisation dans le cas de *retimings* légaux peut permettre de mieux appréhender tous ces problèmes proches et de prévoir leur complexité. En outre, nous montrons dans ce chapitre que des optimisations plus complexes – comme la contraction de tableaux –, mises en œuvre pour la consommation mémoire, peuvent se ramener à un problème de maximisation du nombre d'arcs de poids nul.

D'un point de vue plus pratique, dans un compilateur classique comme *gcc*, une grosse partie des optimisations ne sont faites qu'à bas niveau, c'est-à-dire au niveau de la représentation intermédiaire à trois adresses. La distance de dépendance n'apparaît alors plus explicitement et les dépendances ne sont plus différenciées que par leur appartenance ou non à la même itération. Maximiser au préalable le nombre de dépendances locales peut alors permettre de mieux optimiser le code produit. Mais la compilation peut également être destinée à produire un circuit, comme c'est le cas du compilateur

PICO (voir [98]). Dans ce contexte, un arc de poids nul signifiera un simple fil, alors qu'un arc de poids non nul correspondra à un « FIFO », donc à un surcoût en matériel.

En fait, nous pouvons déjà nous faire une bonne idée sur la complexité de ce problème : tout d'abord, le théorème 4 de la page 119 nous indique que même les parties sans circuits du graphe peuvent poser problème lorsque nous cherchons à rendre le poids de tous les arcs nul. Ensuite, dans le cadre de l'alignement des données, Darte et Robert ont montré dans [23] que le problème de recherche d'un *retiming* quelconque (non légal) maximisant le nombre d'arcs de poids nul du graphe est NP-complet au sens faible. Cependant, l'étude de ce problème reste toujours incomplète, et plusieurs questions se posent encore :

- dans les chapitres 3 et 4, nous avons formulé deux problèmes de *retiming* qui se sont avérés tous deux NP-complets au sens fort. Est-ce également le cas pour le problème de maximisation du nombre d'arcs de poids nul ? Autrement dit, la difficulté vient-elle du côté combinatoire dû aux contraintes de structure du graphe, ou bien le problème est-il un problème de nature numérique admettant une solution pseudo-polynomiale ?
- le problème admet-il une formulation par programmation linéaire comme c'était le cas pour les problèmes de *retiming* des chapitres 3 et 4, ou bien les contraintes sont-elles cette fois-ci non linéaires ?
- le problème reste-t-il NP-complet lorsque nous imposons au *retiming* d'être légal, ou bien cette nouvelle contrainte rend-elle la résolution plus simple ?
- le problème reste-t-il NP-complet pour les graphes sans circuits, ou bien la difficulté ne vient-elle que des circuits ?

Le reste de ce chapitre est organisé de la manière suivante. À la section 5.2 nous montrons que le problème est NP-complet au sens fort, même pour les graphes acycliques. Nous proposons alors en section 5.3 une formulation du problème par programmation linéaire dans les deux cas de figure suivants : *retiming* légal ou *retiming* non légal. À la section 5.4, nous montrons comment nos résultats sur ce problème très simple de maximisation peuvent être directement exploités afin de résoudre un problème réel : la contraction de tableaux. Enfin, nous concluons en section 5.5.

5.2 Maximiser le nombre d'accès locaux est NP-complet

Lorsqu'il n'est pas possible de rendre nul par *retiming* le poids de tous les arcs d'un graphe (à l'aide de l'algorithme 11 présenté à la page 120), est-il possible de trouver un *retiming* tel que G_r ait autant d'arcs de poids nul que possible ? Autrement dit, peut-on maximiser le nombre de dépendances « alignées » de notre graphe ? Une réponse partielle est donnée à cette question dans [23]. Les auteurs prouvent que le problème est NP-complet au sens faible (par réduction d'un problème de partition) sur un graphe quelconque et n'imposent pas au *retiming* d'être légal. Nous prouvons dans cette section que le problème est NP-complet au sens fort et qu'il reste NP-complet même si nous nous restreignons aux *retimings* légaux et aux graphes sans circuits. Par souci de clarté, nous commençons par présenter la preuve pour le cas de graphes quelconques (cycliques ou non), puis nous présentons celle pour le cas de graphes sans circuits. Cette dernière est également valable pour le cas des graphes cycliques (moyennant une très légère modification), néanmoins nous avons préféré présenter les deux preuves car la seconde est plus compliquée et moins intuitive.

5.2.1 Preuve pour le cas général

Dans cette section, nous traitons le cas général de maximisation du nombre d'arcs de poids nul d'un graphe de dépendance. La preuve construit une transformation polynomiale de Not-All-

EQUAL 3SAT (voir [47, p. 259]) en instance de notre problème. Néanmoins, le graphe construit est toujours cyclique. Cette preuve ne s'applique donc plus lorsque nous considérons notre problème de maximisation restreint aux graphes de dépendances sans circuits (cas fréquent dans les programmes réels). Nous présentons la preuve de NP-complétude dans le cas acyclique à la section suivante. À notre problème de maximisation, nous associons le problème de décision suivant :

Problème : MAXIMISATION DES ACCÈS LOCAUX

Instance Un graphe de dépendance $G = (V, E, d)$ et un entier naturel K.

Question Existe-t-il un *retiming* de G tel que G_r ait au moins K arcs de poids nul ?

Nous allons voir que la preuve que nous donnons reste valide si nous imposons au *retiming* d'être légal[1]. Ceci généralise donc la preuve de [23], qui n'est valide que dans le cas de *retimings* quelconques (c'est-à-dire non nécessairement légaux). La preuve est faite par réduction polynomiale du problème NOT-ALL-EQUAL 3SAT (voir le problème LO3 de [47, p. 259]) que nous rappelons ici :

Problème : NOT-ALL-EQUAL 3SAT

Instance Un ensemble U de n variables booléennes et un ensemble C de m clauses sur U (pour chaque variable $u \in U$, u et $\bar{u}$ sont appelés des littéraux sur U, une clause est un ensemble de littéraux sur U) tel que pour toute clause $c \in C$, $|c| = 3$.

Question Existe-t-il un assignement de vérité pour U (c'est-à-dire une fonction $T : U \longrightarrow \{V, F\}$, qui s'étend aux littéraux de la façon suivante : $T(\bar{u}) = V$ si $T(u) = F$ et $T(\bar{u}) = F$ si $T(u) = V$) tel que toute clause $c \in C$ contienne au moins un littéral vrai ($\exists x \in c, T(x) = V$) et un littéral faux ($\exists y \in c, T(y) = F$) ?

Le théorème suivant prouve que MAXIMISATION DES ACCÈS LOCAUX est NP-complet en dimension 1. Notons que nous ne perdons pas en généralité en nous restreignant au cas mono-dimensionnel. En effet, l'extension au cas multi-dimensionnel se fait de la même manière que dans le cas de l'algorithme 11, en utilisant des vecteurs à la place des scalaires (voir section 4.3.2).

Théorème 7 MAXIMISATION DES ACCÈS LOCAUX *est NP-complet au sens fort.*

Preuve Nous commençons par prouver que le problème appartient à NP, puis nous décrivons notre transformation polynomiale d'une instance de NOT-ALL-EQUAL 3SAT en instance de MAXIMISATION DES ACCÈS LOCAUX, enfin nous montrons l'équivalence entre instances positives.

Le problème est dans NP Considérons une instance positive de notre problème : un graphe de dépendance $G = (V, E, d)$ et un entier naturel K, ainsi qu'une solution à cette instance, un *retiming* légal r de G tel que G_r ait au moins K arcs de poids nul. Grâce au lemme 8, nous savons qu'il existe un *retiming* r' de G dont les valeurs sont bornées par une fonction polynomiale des poids de G tel que $G_{r'}$ ait au moins les mêmes arcs de poids nul que G_r. Donc r' est un certificat polynomial de notre instance (sa taille est polynomiale en la taille de l'instance, et vérifier que $G_{r'}$ a K arcs de poids nuls ou plus se fait en temps polynomial : au plus $|E|$ arcs à vérifier).

[1] Nous verrons que nous construisons un graphe pour lequel seul un *retiming* légal peut produire K ou plus arcs de poids nul. Ainsi, toutes les solutions pour le cas de *retimings* quelconques sont également des solutions pour le cas de *retimings* légaux seulement.

Transformation Soit une instance (U, C) de NOT-ALL-EQUAL 3SAT, nous définissons $f(U, C) = (G, K)$ comme une transformation polynomiale de (U, C) en instance de MAXIMISATION DES ACCÈS LOCAUX de la façon suivante :

- nous partons de $G = (V, E)$ avec $V = \emptyset$ et $E = \emptyset$;
- pour chaque variable $u \in U$, nous ajoutons à G deux sommets u et $\bar{u}$, et deux arcs respectivement de u à $\bar{u}$ et de $\bar{u}$ à u, tous deux de poids 1 (voir figure 5.1) ;

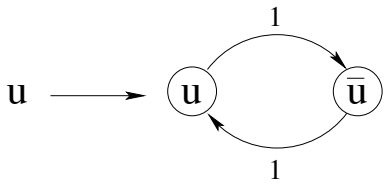


FIG. 5.1 – Transformation d'une variable.

- pour chaque clause $c = \{x, y, z\} \in C$, nous ajoutons à G six arcs de poids 1 respectivement de x à y, de y à x, de x à z, de z à x, de y à z et de z à y (voir figure 5.2) ;

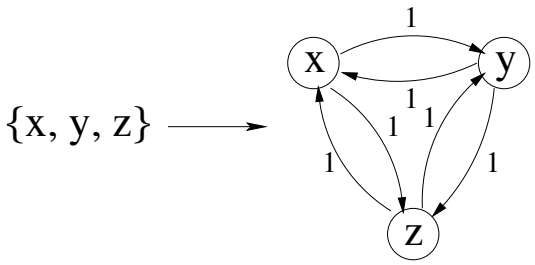


FIG. 5.2 – Transformation d'une clause.

- nous posons $K = 2m + n$ (rappelons que $n = |U|$ et $m = |C|$).

La transformation est clairement polynomiale avec $2|U|$ sommets et $2|U| + 6|C|$ arcs en tout.

Réduction Soit (U, C) une instance de NOT-ALL-EQUAL 3SAT et $(G, K) = f(U, C)$. Nous devons prouver que :

- si (U, C) est une instance positive de NOT-ALL-EQUAL 3SAT, alors G est une instance positive de MAXIMISATION DES ACCÈS LOCAUX. Soit T un assignement de vérité pour U satisfaisant toutes les clauses de C avec au moins un littéral vrai et un littéral faux. Nous définissons, pour toute variable $u \in U$,

$$r(u) = \begin{cases} 1 & \text{si } T(u) = V \\ 0 & \text{si } T(u) = F \end{cases} \qquad r(\bar{u}) = 1 - r(u)$$

  Tout d'abord, remarquons que pour tout arc $e = (u, v) \in E$, $-1 \leq r(v) - r(u)$, donc $w_r(e) = r(v) - r(u) + w(e) \geq 0$ car tous les arcs de G ont un poids égal à 1 (par construction). Donc r est un *retiming* légal de G (et d'autant plus un *retiming* quelconque).

  Pour toute variable $u \in U$, $r(u) - r(\bar{u}) = \pm 1$. Donc soit $w_r((u, \bar{u})) = 0$ et $w_r((\bar{u}, u)) = 2$, soit $w_r((u, \bar{u})) = 2$ et $w_r((\bar{u}, u)) = 0$. Donc chaque construction associée à une variable génère exactement un arc de poids nul, soit n arcs de poids nul pour l'ensemble des variables.

  Pour toute clause $c = \{x, y, z\} \in C$, au moins un littéral est vrai et au moins un autre est faux. Donc il existe deux littéraux, par exemple x et y, tels que $r(x) = r(y)$ et $r(x) - r(z) =$

$r(y) - r(z) = \pm 1$. Donc il n'y a pas d'arc de poids nul entre x et y, et il y en a exactement un entre x et z, tout comme entre y et z. Autrement dit, deux arcs de poids nul dans chaque structure associée à une clause, soit $2m$ arcs de poids nul pour l'ensemble des clauses.

Nous retrouvons donc finalement $2m + n = K$ arcs de poids nul en tout dans G_r, et (G, K) est donc une instance positive de MAXIMISATION DES ACCÈS LOCAUX.

– si $f(U, C) = (G, K)$ est une instance positive de MAXIMISATION DES ACCÈS LOCAUX, alors (U, C) est une instance positive de NOT-ALL-EQUAL 3SAT. Soit r un *retiming* de G tel que G_r ait au moins K arcs de poids nul. Nous définissons, pour tous les littéraux sur U,

$$T(u) = \begin{cases} T & \text{if } r(u) \bmod 2 = 1 \\ V & \text{if } r(u) \bmod 2 = 0 \end{cases}$$

Rappelons que le *retiming* ne change pas le poids d'un circuit, donc au plus un seul des deux arcs associés à une variable peut avoir un poids nul après *retiming* par r (autrement cela signifierait que le poids du circuit est 0 et non 2).

Nous allons alors montrer qu'au plus deux des arcs associés à une clause peuvent avoir un poids nul après *retiming*. Dans un premier temps, la même observation que précédemment nous permet de conclure qu'au plus un seul des deux arcs entre deux littéraux distincts a un poids nul, soit au maximum trois pour l'ensemble de la clause. Supposons alors que, pour une clause $c = \{x, y, z\}$, au moins deux arcs aient un poids nul après *retiming*, par exemple entre x et y (donc $r(y) = r(x) \pm 1$) et entre x et z (donc $r(z) = r(x) \pm 1$). Alors soit $r(y) = r(z)$ soit $r(y) = r(z) \pm 2$, dans les deux cas il n'y a pas d'arc de poids nul entre y et z. Donc il y a au plus deux arcs de poids nul parmi tous ceux générés par une clause (et ce, quel que soit le *retiming*, légal ou non).

En résumé, G_r contient au plus $2m + n$ arcs de poids nul, et par hypothèse il en contient au moins $K = 2m + n$. Donc G_r contient exactement $K = 2m + n$ arcs de poids nul, donc un pour chaque groupe d'arcs associés à une variable, et deux pour chaque groupe d'arcs associés à une clause.

Il reste à montrer que T est un assignement de vérité et qu'il satisfait toutes les clauses avec au plus deux littéraux vrais dans chaque clause.
- puisqu'il existe un arc de poids nul entre u et $\bar{u}$, nous avons $r(u) = r(\bar{u}) \pm 1$ (autrement les deux arcs seraient de poids non nul). Donc $r(u) \bmod 2 \neq r(\bar{u}) \bmod 2$ et $T(u) \neq T(\bar{u})$;
- chaque clause $c = \{x, y, z\}$ contient exactement deux arcs de poids nul ; considérons l'un d'entre eux, par exemple entre x et y. Comme l'arc choisi est nul, nous avons $r(x) = r(y) \pm 1$ et donc $T(x) \neq T(y)$.

Donc T est bien l'assignement de vérité cherché et (U, C) est une instance positive de NOT-ALL-EQUAL 3SAT.

Ceci termine la preuve : MAXIMISATION DES ACCÈS LOCAUX est NP-complet au sens fort. □

Comme nous l'avons déjà précisé, nous pouvons remarquer que la même preuve tient, que nous imposions au *retiming* d'être légal ou non. En effet, dans la première partie de la preuve, si (U, C) est une instance positive de NOT-ALL-EQUAL 3SAT, nous construisons un *retiming* légal de G, et dans la seconde partie, s'il existe un *retiming* r de G tel que G_r ait K arcs de poids nuls ou plus, alors, même s'il est légal, le raisonnement tient toujours.

5.2.2 Preuve pour le cas acyclique

Dans cette section, nous reprenons le problème de la section 5.2.1 en nous limitant au cas de graphes sans circuits. Autrement dit, étant donné un graphe sans circuits G, nous souhaitons trouver un *retiming* r de G tel que G_r ait le plus possible d'arcs de poids nul. Nous appelons ce problème MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX et nous définissons le problème de décision associé de la manière suivante :

Problème : MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX

Instance Un graphe de dépendance acyclique $G = (V, E, w)$ et un entier naturel K.

Question Existe-t-il un *retiming* (éventuellement légal) de G tel que G_r ait au moins K arcs de poids nul ?

La preuve de NP-complétude de ce problème est fondée sur une réduction polynomiale du problème ONE-IN-THREE 3SAT (voir le problème LO4 de [47, p. 259]) dont nous rappelons ici la définition :

Problème : ONE-IN-THREE 3SAT

Instance Un ensemble U de n variables booléennes et un ensemble C de m clauses sur U (pour chaque variable $u \in U$, u et $\bar{u}$ sont appelés des littéraux sur U, une clause est un ensemble de littéraux sur U) tel que pour toute clause $c \in C$, $|c| = 3$.

Question Existe-t-il un assignement de vérité pour U (c'est-à-dire une fonction $T : U \longrightarrow \{V, F\}$, qui s'étend au littéraux de la façon suivante : $T(\bar{u}) = V$ si $T(u) = F$ et $T(\bar{u}) = F$ si $T(u) = V$) tel que toute clause $c \in C$ contienne exactement un littéral vrai ($\exists! x \in c, T(x) = V$) ?

De façon analogue au cas général, le théorème suivant prouve que MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX est NP-complet au sens fort en dimension 1. Ici encore, l'extension au cas multi-dimensionnel se fait de la même manière que dans le cas de l'algorithme 11 (voir section 4.3.2)

Théorème 8 MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX *est NP-complet au sens fort.*

Preuve Nous commençons par prouver que le problème appartient à NP, puis nous décrivons notre transformation polynomiale d'une instance de ONE-IN-THREE 3SAT en instance de MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX, enfin nous montrons l'équivalence entre instances positives.

Le problème est dans NP Même preuve que dans le cas général.

Transformation Soit une instance (U, C) de ONE-IN-THREE 3SAT, nous définissons $f(U, C) = (G, K)$ comme une transformation polynomiale de (U, C) en instance de MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX de la façon suivante :

- nous partons d'un graphe $G = (V, E, w)$ contenant deux sommets a et b ($V = \{a, b\}$) et où E est un ensemble de $48mn + 8m$ arcs de poids nul dirigés de a vers b. Nous appelons cette structure initiale la structure de base de G (voir figure 5.3) ;
- pour chaque variable $u \in U$, nous ajoutons à G deux sommets u et $\bar{u}$, ainsi que $24m$ arcs de poids 1 de $\bar{u}$ vers b, $16m$ arcs de poids 1 de u vers $\bar{u}$, $24m$ arcs de poids 0 de a vers u, $16m$ arcs de poids 0 de a vers $\bar{u}$ et $16m$ arcs de poids 1 de u vers b (voir figure 5.4) ;

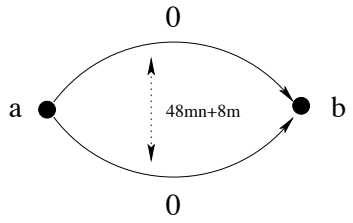


FIG. 5.3 – Point de départ de la transformation (structure de base).

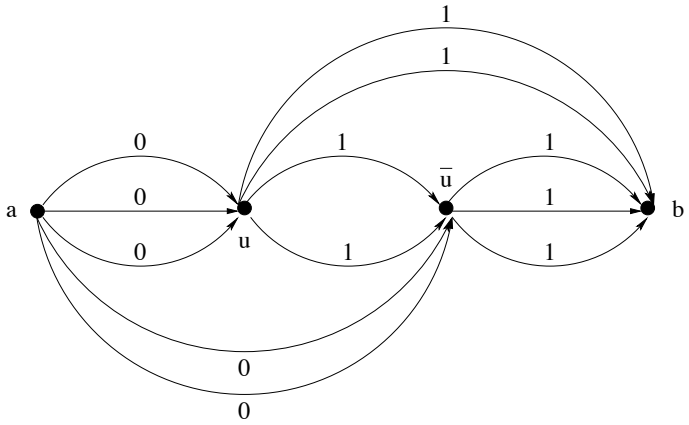


FIG. 5.4 – Transformation d'une variable (chaque arc représenté sur la figure est répété $8m$ fois).

– pour chaque clause $c = \{x, y, z\} \in C$, nous considérons ce que nous appelons des clauses dérivées $c_0 = \{x, \bar{y}, \bar{z}\}$, $c_1 = \{\bar{x}, y, \bar{z}\}$ et $c_2 = \{\bar{x}, \bar{y}, z\}$. Nous ajoutons à G trois sommets c_0, c_1 et c_2, puis pour chaque clause dérivée $c_i = \{x', y', z'\}$, $i \in \{0, 1, 2\}$, un arc de poids 0 de a vers c_i, un arc de poids 0 de c_i vers chacun des sommets x', y', et z' (construits par les variables) et un arc de poids 1 de chacun des sommets x', y', et z' vers b (voir figure 5.5) ;

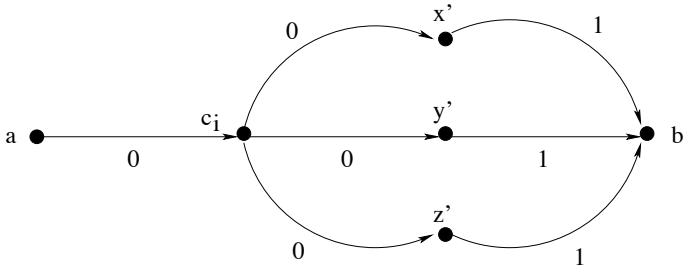


FIG. 5.5 – Transformation d'une clause (pour une seule des clauses dérivées).

– nous posons $K = 96mn + 22m$.

Le graphe G généré de cette manière est clairement sans circuit. La transformation est polynomiale car G a $2n + 3m + 2$ sommets et $144mn + 29m$ arcs.

Réduction Soit (U, C) une instance de ONE-IN-THREE 3SAT et $(G, K) = f(U, C)$. Nous devons prouver que :

– si (U, C) est une instance positive de ONE-IN-THREE 3SAT, alors G est une instance positive de MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX. Soit T un assignement de vérité pour

U satisfaisant toutes les clauses de C avec exactement un littéral vrai. Nous définissons r *retiming* de G de la manière suivante :

$$r(a) = r(b) = 0$$

$$\forall u \in U,$$
$$r(u) = \begin{cases} 1 & \text{si } T(u) = V \\ 0 & \text{sinon} \end{cases} \qquad r(\bar{u}) = 1 - r(u)$$

$$\forall c \in C,\ \forall c_i = \{x', y', z'\},\ i \in \{0, 1, 2\},$$
$$r(c_i) = \begin{cases} 1 & \text{si } T(x') = T(y') = T(z') = V \\ 0 & \text{sinon} \end{cases}$$

Puisque $r(a) = r(b)$, le *retiming* est légal pour la structure de base de G et les $48mn + 8m$ arcs associés restent de poids nul.

Selon T, pour chaque variable $u \in U$, nous avons soit $r(u) = 1$ et $r(\bar{u}) = 0$, soit $r(u) = 0$ et $r(\bar{u}) = 1$. Dans les deux cas, cela donne naissance à exactement six groupes de $8m$ arcs de poids nul dans chaque construction associée à une variable (soit six arcs de poids nul sur la figure 5.4). Donc, un total de $48mn$ arcs de poids nul pour l'ensemble des variables (et notons au passage que le *retiming* est légal pour tous les arcs associés aux variables).

Pour chaque clause $c \in C$, selon T, exactement un des littéraux de c est vrai, donc exactement une seule $c_i = \{x', y', z'\}$ des clauses dérivées $\{c_0, c_1, c_2\}$ a tous ses littéraux vrais pour T. Pour cette clause dérivée, nous avons $r(x') = r(y') = r(z') = 1$ et $r(c_i) = 1$, donc le *retiming* est légal pour la construction associée à c_i et génère six arcs de poids nul. Pour toutes les autres clauses dérivées c_j, $r(c_j) = 0$ puisqu'au moins l'un des littéraux qui la composent est faux, et quelle que soit la valeur de *retiming* des autres sommets (0 ou 1), le *retiming* reste légal pour la construction associée à c_j et génère quatre arcs de poids nul. Donc au total $6 + 4 + 4 = 14$ arcs de poids nul par clause complète et $14m$ arcs de poids nul pour l'ensemble des clauses.

Pour l'ensemble du graphe, nous obtenons donc $96mn + 22m$ arcs de poids nul : (G, K) est une instance positive de MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX.

– si $f(U, C) = (G, K)$ est une instance positive de MAXIMISATION ACYCLIQUE DES ACCÈS LOCAUX alors (U, C) est une instance positive de ONE-IN-THREE 3SAT. Soit r un *retiming* de G tel que G_r ait au moins K arcs de poids nul. Sans perte de généralité, nous pouvons supposer que $r(a) = 0$, autrement nous soustrayons $r(a)$ à toutes les valeurs de *retiming* des sommets de G. Nous définissons, pour tous les littéraux sur U,

$$\forall u \in U,\ T(u) = \begin{cases} F & \text{si } r(u) = 0 \\ V & \text{sinon} \end{cases} \quad \text{et } T(\bar{u}) = \begin{cases} F & \text{si } r(\bar{u}) = 0 \\ V & \text{sinon} \end{cases}$$

Avant d'aller plus loin, nous nous proposons de prouver un certain nombre de propriétés de tout *retiming* satisfaisant notre instance (G, K). Nous prouvons tout d'abord que $r(a) = r(b) = 0$, puis que, pour toute variable $u \in U$, $r(u)$ et $r(\bar{u})$ ont pour valeur soit 0 soit 1.

Nous pouvons remarquer que chaque structure construite à partir d'une variable contient $12 * 8m = 96m$ arcs et que chaque structure construite à partir d'une clause contient $3 * 7 = 21$ arcs, soit un total de $96mn + 21m$ arcs pour l'ensemble de ces structures. En conséquence, quel que soit le *retiming* r choisi, si G_r contient $K = 96mn + 22m$ arcs de poids nul ou plus, alors certains d'entre eux appartiennent à la structure de base. Puisque tous les arcs de cette

structure ont le même poids (0), la même source (a) et le même puits (b), tous ont donc un poids nul dans G_r et $r(a) = r(b) = 0$. Il y a donc exactement $48mn + 8m$ arcs de poids nul dans la structure de base.

Considérons maintenant la structure construite à partir d'une variable $u \in U$. Supposons que $r(u) < 0$ ou $r(u) > 1$: dans les deux cas, les arcs de a vers u et les arcs de u vers b ont un poids non nul après *retiming* (puisque $r(a) = r(b) = 0$). De plus, les arcs de a vers $\bar{u}$ et de u vers $\bar{u}$ ne peuvent pas avoir un poids nul simultanément (autrement, nous aurions $r(\bar{u}) = 0$ et $r(u) = 1$). Donc, au plus $40m$ arcs ont un poids nul après *retiming*, les arcs de a vers $\bar{u}$ ou bien les arcs de u vers $\bar{u}$, plus éventuellement les arcs de $\bar{u}$ vers b. Supposons maintenant que $r(\bar{u}) < 0$ ou $r(\bar{u}) > 1$: dans les deux cas, les arcs de a vers $\bar{u}$ et les arcs de $\bar{u}$ vers b ont un poids non nul après *retiming* (puisque $r(a) = r(b) = 0$). De plus, les arcs de u vers $\bar{u}$ et les arcs de u vers b ne peuvent pas avoir un poids nul simultanément car ils ont la même source (u), le même poids initial (1), mais $r(\bar{u}) \neq r(b)$. Donc, là encore, la structure contient au plus $40m$ arcs de poids nul après *retiming*, les arcs de u vers $\bar{u}$ ou les arcs de u vers b, plus éventuellement les arcs de a vers u. Enfin, nous pouvons facilement vérifier que si $r(u) = r(\bar{u}) = 0$ ou si $r(u) = r(\bar{u}) = 1$, alors, après *retiming*, exactement $40m$ arcs de poids nul sont dans la structure construite à partir de u, et si $r(u) = 0$ et $r(\bar{u}) = 1$ ou si $r(u) = 1$ et $r(\bar{u}) = 0$, alors, après *retiming*, exactement $48m$ arcs de poids nul sont dans la structure construite à partir de u.

Supposons à présent qu'au moins une des structures générées à partir des variables ait $40m$ arcs de poids nul ou moins après *retiming*. Alors le nombre total d'arcs de poids nul contenus dans l'ensemble des structures associées aux variables de U plus la structure de base est au plus $48mn + 8m + 48m(n-1) + 40m = 96mn$. Par choix de K, il nous faut encore $22m$ arcs de poids nul dans les structures associées aux clauses pour satisfaire l'instance, pourtant le nombre total d'arcs restant dans ces structures est de $21m$: cela est donc impossible. Donc, puisque r est une solution de notre instance, toutes les structures générées à partir des variables contiennent exactement $48m$ arcs de poids nul après *retiming* et, pour chaque variable $u \in U$, soit $r(u) = 0$ et $r(\bar{u}) = 1$, soit $r(u) = 1$ et $r(\bar{u}) = 0$ (notons au passage que le *retiming* est alors légal pour les arcs de la structure). Il s'ensuit que pour toute variable $u \in U$, $T(u) \neq T(\bar{u})$ (par définition de T) et donc T est un assignement de vérité. Il reste à prouver que T satisfait toutes les clauses de C avec exactement un littéral vrai.

Considérons une clause dérivée $c_i = \{x', y', z'\}$ obtenue à partir d'une clause $c \in C$: elle correspond à une structure dans laquelle $r(x')$, $r(y')$ et $r(z')$ ont pour valeur soit 0 soit 1 (puisque chacun de ces sommets est le u ou le $\bar{u}$ contenu dans la structure associée à la variable u). Supposons que $r(c_i) < 0$ ou $r(c_i) > 1$, alors l'arc de a vers c_i et les arcs de c_i vers chacun des x', y' et z' ont un poids non nul après *retiming*. Ainsi, dans ce cas, au plus 3 arcs ont un poids nul après *retiming*. De plus, si $r(c_i) = 0$, alors la structure contient exactement 4 arcs de poids nul après *retiming*, et si $r(c_i) = 1$, alors soit $r(x') = r(y') = r(z') = 1$ et la structure contient exactement 6 arcs de poids nul après *retiming*, soit il est facile de vérifier que la structure n'en contient que 4 au plus (avec certains de poids négatif).

Enfin, par construction des clauses dérivées c_0, c_1 et c_2, pour toute paire de telles clauses (c_i, c_j), il existe une variable $u \in U$ telle que $u \in c_i$ et $\bar{u} \in c_j$ (ou inversement). Donc, au plus une seule des trois structures associées aux clauses dérivées vérifie $r(x') = r(y') = r(z') = 1$ et contient 6 arcs de poids nul (toutes les autres contiennent au plus 4 arcs de poids nul). Supposons alors que pour l'une des clauses il y ait moins de $6+4+4$ arcs de poids nul générés dans l'ensemble de ses clauses dérivées après *retiming*. En les ajoutant aux arcs des autres

clauses, nous obtenons au plus $(6+4+4)*(m-1)+13 = 14m-1$ arcs de poids nul. En ajoutant ce total à celui des autres structures, nous obtenons au plus $48mn+8m+48mn+14m-1 = 96mn+22m-1$ arcs de poids nul et le *retiming* n'est alors pas une solution de notre instance. Donc, pour toutes les clauses $c \in C$, au moins l'une de ses clauses dérivées $c_i = \{x', y', z'\}$ contient 6 arcs de poids nul et les deux autres 4. Ceci signifie que $r(x') = r(y') = r(z') = 1$ et, par définition de T, que tous les littéraux de c_i sont vrais. Donc, par définition de c_i, cela signifie qu'exactement un seul des littéraux de c est vrai.

Donc T est un assignement de vérité satisfaisant toutes les clauses avec exactement un littéral vrai : (U, C) est une instance positive de One-in-Three 3SAT.

Ceci termine la preuve : Maximisation acyclique des accès locaux est NP-complet au sens fort. □

Comme nous l'avons déjà précisé, nous pouvons remarquer que la même preuve tient, que nous imposions au *retiming* d'être légal ou non. En effet, dans la première partie de la preuve, si (U, C) est une instance positive de One-in-Three 3SAT, nous construisons un *retiming* légal de G, et dans la seconde partie, s'il existe un *retiming* r de G tel que G_r ait K arcs de poids nuls ou plus, alors, même s'il est légal, le raisonnement est toujours valable.

Par ailleurs, le graphe que nous construisons est un multigraphe (plusieurs arcs distincts peuvent exister entre deux mêmes sommets). Si nous souhaitons interdire les multigraphes, nous pouvons ajouter au milieu de chaque arc créé un nouveau sommet (qui coupe l'arc en deux arcs distincts). Afin de ne pas surcharger ce document nous ne détaillerons pas cette variante de la preuve. Cependant le poids des nouveaux chemins créés (qui passent chacun par un des nouveaux sommets) varie de la même façon que le poids des arcs qu'ils remplacent. En choisissant de manière appropriée la valeur de *retiming* des nouveaux sommets nous pouvons rendre toujours nul leur arc sortant. La preuve est alors la même (en comptant les quelques arcs de poids nul supplémentaires sortant des nouveaux sommets).

5.3 Solution par programmation linéaire

Dans cette section, nous proposons une solution par programmation linéaire en nombres entiers au problème de la maximisation du nombre d'arcs de poids nul par *retiming*. Comme nous l'avons vu en section 5.2, le problème est NP-complet, par conséquent la complexité de notre solution sera *a priori* non polynomiale. Cependant, comme nous le verrons, le programme est composé d'un faible nombre de contraintes, ce qui rend sa résolution envisageable en pratique.

Proposition 16 *Soit un graphe de dépendance $G = (V, E, d)$ et $M = \sum_{e \in E} |d(e)|$. Toute solution optimale (r, k) du programme linéaire suivant est telle que r est un* retiming *maximisant le nombre d'arcs de poids nul de G.*

$$\begin{array}{ll} \text{Minimiser :} & \\ & \sum_{e \in E} k(e) \\ \text{Tel que :} & \\ \forall e = (u, v) \in E, & Mk(e) \geq d(e) + r(v) - r(u) \\ & Mk(e) \geq -(d(e) + r(v) - r(u)) \end{array} \tag{5.1}$$

Preuve Remarquons tout d'abord que, dans ce système d'inéquations, les deux inégalités sont équivalentes à l'inégalité suivante :

$$Mk(e) \geq |d_r(e)|$$

Cette dernière forme est certainement plus simple à comprendre intuitivement : si le poids après *retiming* n'est pas nul, alors $k(e)$ devra être strictement positif.

Soit une solution optimale (r, k) de 5.1 et $K = |E| - \sum_{e \in E} k(e)$. Le coût de notre programme linéaire est alors égal à $|E| - K$. Nous pouvons supposer $M > 0$, dans le cas contraire tous les arcs ont déjà un poids nul. Donc, d'après les contraintes, nous avons $d(e) + r(v) - r(u) = 0 \Rightarrow k(e) \geq 0$ et $d(e) + r(v) - r(u) \neq 0 \Rightarrow k(e) \geq 1$. À cause de $d_r(e) \neq 0 \Rightarrow k(e) \geq 1$, au moins K arcs $e \in E$ sont tels que $d(e) + r(v) - r(u) = 0$, autrement le nombre d'arcs de poids non nul de G_r serait supérieur ou égal à $|E| - K + 1$, et le coût de (r, k) serait au moins de $|E| - K + 1$.

Supposons alors que r ne maximise pas le nombre d'arcs de poids nul du graphe. Alors il existe r_2 tel que G_{r_2} ait $K' \geq K + 1$ arcs de poids nul. D'après le lemme 8 de la page 132, il existe r' tel que tous les arcs de poids nul de G_{r_2} ont également un poids nul dans $G_{r'}$ et $\forall v \in V$, $0 \leq r'(v) \leq \sum_{\{e \in E | d_{r_2}(e) = 0\}} |d(e)|$. Donc, soit $d(e) + r_2(v) - r_2(u) = 0$ et alors $d(e) + r'(v) - r'(u) = 0$, soit $d(e) + r_2(v) - r_2(u) \neq 0$, donc e n'apparaît pas dans la somme $\sum_{\{e \in E | d_{r_2}(e) = 0\}} |d(e)|$ et $-\sum_{\{e \in E\}} |d(e)| \leq d(e) + r'(v) - r'(u) \leq \sum_{\{e \in E\}} |d(e)|$. Dans tous les cas $-M \leq d(e) + r'(v) - r'(u) \leq M$, donc en choisissant $k'(e)$ tel que $k'(e) = 0$ si $d(e) + r'(v) - r'(u) = 0$ et $k'(e) = 1$ si $d(e) + r'(v) - r'(u) \neq 0$, nous obtenons une solution admissible (r', k') de coût $|E| - K' \leq |E| - K - 1$, ce qui contredit l'optimalité de (r, k). □

Évidemment, comme nous pouvons le voir, ce programme est destiné à produire un *retiming* sans contraintes de légalité, répondant aux besoins des problèmes d'alignement [23] mais non aux besoins des problèmes de transformation de programme pour la localité. Nous proposons donc également la version restreinte aux *retimings* légaux. Auparavant, nous souhaitons néanmoins présenter une ultime variante du lemme 7 de la page 126 simplement destinée à simplifier l'expression de la constante M du programme linéaire qui va suivre (nous parvenons au même résultat en utilisant le lemme 7 mais avec une constante M plus grande). Ici, nous reprenons la condition du lemme 8 dans le cas d'un *retiming* légal.

Lemme 9 *Soit un graphe de dépendance $G = (V, E, d)$. Pour tout* retiming *légal r de G, il existe un* retiming *légal r' de G tel que $d_r(e) = 0 \Rightarrow d_{r'}(e) = 0$ (tous les arcs de poids nul dans G_r ont également un poids nul dans $G_{r'}$) et tel que $\forall v \in V$, $0 \leq r'(v) \leq \sum_{\{e \in E | d(e) < 0 \text{ ou } d_r(e) = 0\}} |d(e)|$.*

Preuve La preuve est analogue à celle du lemme 7. Nous construisons un graphe $G' = (V, E', d')$ à partir de G et r de la manière suivante : pour tout arc $e = (u, v) \in E$,

- si e est tel que $d_r(e) = 0$, nous posons $d'(e) = d(e)$ et nous ajoutons à E' un arc $e' = (v, u)$ de poids $d'(e') = -d(e)$. Notons qu'alors e et e' forment un circuit de poids nul ;
- sinon, $d_r(e) \geq 1$ et nous posons $d'(e) = d(e)$.

Par construction, G'_r n'a que des arcs de poids positif ou nul, donc r est légal pour G' et par le corollaire 1 G' ne possède pas de circuit de poids strictement négatif. Nous pouvons donc trouver à l'aide de l'algorithme 2 un nouveau *retiming* r' de G' défini à partir d'un plus court chemin dans G'. Défini de cette façon, r' ne possède que des valeurs positives, et par construction de G' ce nouveau *retiming* rend nuls au moins les mêmes arcs que r. Enfin, comme l'algorithme 2 définit la valeur de *retiming* comme l'opposé d'un plus court chemin dans G', $\forall v \in V$, $0 \leq r'(v) \leq \sum_{\{e \in E | d'(e) < 0\}} |d'(e)| = \sum_{\{e \in E | d(e) < 0 \text{ ou } d_r(e) = 0\}} |d(e)|$. □

Nous pouvons donc maintenant proposer un programme linéaire pour la maximisation du nombre d'arcs de poids nul par *retiming* légal cette fois-ci.

Proposition 17 *Soit un graphe de dépendance $G = (V, E, d)$ et $M = \sum_{e \in E} |d(e)|$. Toute solution optimale (r, k) du programme linéaire suivant est telle que r est un* retiming *légal maximisant le*

nombre d'arcs de poids nul de G.

$$\begin{array}{ll} \text{Minimiser :} & \\ & \sum_{e\in E} k(e) \\ \text{Tel que :} & \\ \forall e=(u,v)\in E, & Mk(e) \geq d(e)+r(v)-r(u) \\ & d(e)+r(v)-r(u)\geq 0 \end{array} \tag{5.2}$$

Preuve La preuve est très similaire à celle de la proposition 16. Soit une solution optimale (r,k) de 5.2 et $K=|E|-\sum_{e\in E}k(e)$. Le coût de notre programme linéaire est alors égal à $|E|-K$. Nous pouvons supposer $M>0$, dans le cas contraire tous les arcs ont déjà un poids nul. Donc, d'après les contraintes, nous avons $d(e)+r(v)-r(u)=0 \Rightarrow k(e)\geq 0$ et $d(e)+r(v)-r(u)>0 \Rightarrow k(e)\geq 1$. À cause de $d_r(e)>0 \Rightarrow k(e)\geq 1$, au moins K arcs $e\in E$ sont tels que $d(e)+r(v)-r(u)=0$, autrement le nombre d'arcs de poids non nul de G_r serait supérieur ou égal à $|E|-K+1$, et le coût de (r,k) serait au moins de $|E|-K+1$.

Supposons alors que r ne maximise pas le nombre d'arcs de poids nul du graphe, alors il existe un *retiming* légal r_2 tel que G_{r_2} ait $K'\geq K+1$ arcs de poids nul. D'après le lemme 9, il existe r' légal, tel que tous les arcs de poids nul dans G_{r_2} ont également un poids nul dans $G_{r'}$ et $\forall v\in V, 0\leq r'(v)\leq \sum_{\{e\in E|d(e)<0 \text{ ou } d_{r_2}(e)=0\}}|d(e)|$. Donc, si $d(e)+r_2(v)-r_2(u)=0$ alors $d(e)+r'(v)-r'(u)=0$, sinon $0\leq d(e)+r'(v)-r'(u)\leq d(e)+\sum_{\{e\in E|d(e)<0 \text{ ou } d_{r_2}(e)=0\}}|d(e)|\leq \sum_{\{e\in E\}}|d(e)|$, car soit $d(e)>0$ et il n'apparaît pas dans la somme $\sum_{\{e\in E|d(e)<0 \text{ ou } d_{r_2}(e)=0\}}|d(e)|$, soit $d(e)\leq 0$ et l'inégalité tient. Dans tous les cas, $0\leq d(e)+r'(v)-r'(u)\leq M$, donc en choisissant $k'(e)$ tel que $k'(e)=0$, si $d(e)+r'(v)-r'(u)=0$ et $k'(e)=1$ si $d(e)+r'(v)-r'(u)>0$, nous obtenons une solution admissible (r',k') de coût $|E|-K'\leq |E|-K-1$, ce qui contredit l'optimalité de (r,k). □

5.4 Application à la contraction de tableaux

Dans cette section, nous montrons avec le problème de contraction de tableaux comment un objectif complexe d'optimisation peut se ramener à notre problème simple de maximisation du nombre d'arcs de poids nul. La contraction de tableaux est une technique bien connue en compilation ([72, 75, 45, 31, 78, 46]). Elle consiste à rendre locaux à une même itération d'une boucle tous les accès à chaque élément d'un tableau. Le but est de pouvoir remplacer le tableau en question par un simple scalaire. Plus précisément, si pour chaque opération accédant à un des éléments du tableau en écriture tous les accès en lecture dépendant de cette écriture se font dans la même itération, nous dirons que l'utilisation du tableau est locale. Si l'utilisation d'un tableau est locale et si ce tableau n'est plus utilisé (en lecture) une fois la boucle terminée, alors il peut être simplement remplacé par un scalaire. Ce remplacement est appelé contraction. Nous dirons qu'un tableau est contractable lorque son utilisation est locale. La compilation de programmes écrits dans des langages proposant des opérations sur des tableaux (comme HPF ou FORTRAN 90) donne souvent lieu à l'apparition de tableaux temporaires qui sont des candidats particulièrement favorables à la contraction ([46, 72]). En outre, dans un contexte de data-parallélisme (par exemple avec un langage comme HPF [66]), un tableau dont l'utilisation est locale est également dit privatisable (voir [73, 103]). Un tableau privatisable, même s'il n'est pas contractable en scalaire (par exemple, à cause d'utilisations en dehors de la boucle), est intéressant car il ne génère aucune communication lors de son utilisation. Notons que la privatisation est supportée de manière native dans HPF [66].

En termes de graphe, l'utilisation locale d'un tableau se traduit par le fait que toutes les dépendances de flot sortant des sommets associés aux écritures dans le tableau doivent avoir un poids nul

et concerner des opérations de la même boucle. Dans le cas contraire, cela signifie qu'un élément est défini dans une itération précédant sa lecture ou bien dans une boucle précédente et par conséquent que le tableau n'est pas contractable selon notre definition[2]. La figure 5.6 représente un exemple simple pour lequel un décalage permet de contracter un tableau (cet exemple est minimal puisque le graphe associé ne comporte qu'un seul arc). Soit N un nid de boucles (non nécessairement parfait),

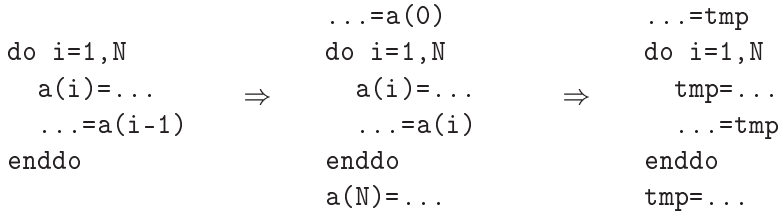


FIG. 5.6 – Exemple de tableau contractable après *retiming*.

et T l'ensemble des tableaux n'étant plus utilisés en lecture après l'exécution du nid N. Pour un graphe $G = (V, E, d)$, nous définissons $E_{flot} \subset E$ l'ensemble des dépendances de flot contenues dans E. Pour tout élément $x \in T$, notons $def(x)$ l'ensemble des éléments de V accédant au tableau x en écriture. D'un point de vue général, notre but est de trouver un *retiming* et une partition des sommets (en une séquence de boucles) qui maximise le nombre d'éléments de T dont l'utilisation est locale et dont tous les sommets associés appartiennent à la même boucle. Autrement dit, nous envisageons la possibilité de produire une solution distribuée (c'est-à-dire sous la forme d'une séquence de boucles). Une solution distribuée est définie par une partition des sommets du graphe de dépendance du nid de boucles. Nous pouvons associer à une telle partition un graphe défini de la façon suivante.

Définition 24 (Sous-graphe induit par une partition) *Soit un graphe de dépendance $G = (V, E, d)$, et $P = \{n_1, \ldots, n_k\}$ une partition de V. Nous définissons le sous graphe induit par P, $G_P = (P, E_P, d_P)$ le graphe dont les sommets sont les éléments de P et dont les arcs et leur poids sont définis de la façon suivante :*

- $E_P = \{(n_i, n_j) \in P^2 \mid n_i \neq n_j \text{ et } \exists (u, v) \in E, (u \in n_i \text{ et } v \in n_j)\}$;
- $\forall e_P = (n_i, n_j) \in E_P, d(e_P) = \min\{d(e) \mid e = (u, v) \in E, u \in n_i \text{ et } v \in n_j\}$.

Une partition des sommets du graphe de dépendance définira une distibution valide d'une part s'il est possible de trouver un ordre total sur la partition tel qu'il n'y ait aucune dépendance d'un élément de la partition vers un élément inférieur (autrement dit aucune boucle ne dépend d'un calcul effectué par une boucle ultérieure) et, d'autre part, si toutes les dépendances incluses dans un élément de la partition (donc dans une même boucle) ont un poids positif ou nul. Ceci est formalisé par la définition suivante.

Définition 25 (Distribution légale) *Soit $G = (V, E, d)$ un graphe de dépendance. Une distribution légale est une partition P des sommets de G telle que G_P, le graphe induit par P, soit sans*

[2]Remarquons toutefois que notre définition d'un tableau contractable ne recouvre pas tous les cas pour lesquels un tableau peut être remplacé par un scalaire : de façon plus générale, nous pouvons remplacer un tableau par un scalaire si, pour chaque instruction écrivant dans un de ses éléments, le nombre maximal de valeurs en vie associé à cette écriture est inférieur ou égal à 1. Ce nombre maximal de valeurs en vie est défini de la même manière qu'au chapitre 3. Il fait donc intervenir l'ordonnancement du corps de la boucle et augmente en conséquence la difficulté du problème. Par ailleurs, notons que les définitions de contractabilité d'un tableau que nous retrouvons dans la littérature sont analogues à la notre. Nous nous contenterons de notre définition bien qu'elle soit légèrement trop forte pour capturer de façon complète la notion de contractabilité.

circuit et

$$\forall e = (u, v) \in E, (\exists n_i \in P, u \in n_i \wedge v \in n_i) \Rightarrow d(e) \geq 0$$

Notre problème de contraction s'exprime alors de la manière suivante : soit un graphe de dépendance $G = (V, E, d)$, nous cherchons un *retiming* r de G et une distribution légale P de G_r tels que le nombre de tableaux contractables :

$$|\{x \in T \mid \forall u \in def(x), \forall e = (u, v) \in E_{flot}, (d(e) + r(v) - r(u) = 0) \wedge (\exists n_i \in P, u \in n_i \wedge v \in n_i)\}|$$

soit maximal (où un tableau est contractable si pour toutes les opérations écrivant dans un de ses éléments alors toutes les opérations dépendantes de cette écriture sont dans la même itération et dans la même boucle). Pour tout tableau contractable, il est alors possible de remplacer son expression par un scalaire dans chaque instruction d'écriture ainsi que dans toutes les instructions qui en dépendent (un scalaire distinct pour chaque instruction d'écriture). À vrai dire, considérer une distribution quelconque n'est pas réellement nécessaire puisque la proposition suivante montre que nous pouvons toujours déduire une solution totalement fusionnée (c'est-à-dire une partition réduite à un seul ensemble) d'une solution distribuée lorsque les dépendances sont uniformes :

Proposition 18 *Soit un graphe de dépendance $G = (V, E, d)$ dont les dépendances sont uniformes. Pour toute solution distribuée (r, P) au problème de contraction de tableaux dans G constituée d'un* retiming *r quelconque de G et d'une distribution légale P de G_r, il existe un* retiming *légal r' tel que la boucle totalement fusionnée associée a au moins le même nombre de tableaux contractables que la séquence produite par (r, P).*

Preuve Comme P est une distribution légale de G_r, $(G_r)_P$ est acyclique, il existe donc un *retiming* légal r_P de $(G_r)_P$. Soit r_1 l'extension de r_P à G défini de la manière suivante : pour tout sommet $u \in V$ de G, il existe un unique $n_i \in P$ tel que $u \in n_i$, nous posons alors $r_1(u) = r_P(n_i)$. Nous définissons $r' = r + r_1$; il nous reste à montrer que r' est un *retiming* légal de G et que tous les tableaux contractables pour (r, P) le sont également pour r' (avec l'ensemble de tous les sommets comme unique élément de la partition). Pour tout arc $e = (u, v) \in E$, $d_{r'}(e) = d(e) + r'(v) - r'(u) = d(e) + r(v) + r_1(v) - r(u) - r_1(u) = d_r(e) + r_1(v) - r_1(u)$. S'il existe $n_i \in P$ tel que $u \in n_i$ et $v \in n_i$, alors $r_1(u) = r_1(v)$ (par construction), donc $d_{r'}(e) = d_r(e) \geq 0$ par légalité de la distribution P des sommets de G_r. Dans le cas contraire, il existe $n_i \in P$ et $n_j \in P$ tels que $u \in n_i$ et $v \in n_j$ où $n_i \neq n_j$. Nous avons alors $d_{r'}(e) = d_r(e) + r_1(v) - r_1(u) \geq (d_r)_P(n_i, n_j) + r_P(n_j) - r_P(n_i) \geq 0$ par construction de $(G_r)_P$, par définition de r_1 et par légalité de r_P. Donc dans tous les cas $d_{r'}(e) \geq 0$ et r' est un *retiming* légal de G et peut donc être écrit sous la forme d'une simple boucle totalement fusionnée. De plus, pour tout tableau contractable $x \in T$ avec la solution (r, P), nous avons $\forall u \in def(x), \forall e = (u, v) \in E_{flot}, \exists n_i \in P, u \in n_i \wedge v \in n_i$ par définition de la contractabilité. Dans ce cas, comme nous l'avons vu, $d_{r'}(e) = d_r(e) = 0$ par contractabilité de x dans (r, P). Donc x vérifie également les conditions de contractabilité pour r' avec une fusion totale (tous les sommets sont dans la même boucle et toutes les dépendances de flot sortantes dans la même itération). □

Ainsi, sans perte de généralité, nous pouvons nous limiter à la recherche d'un *retiming* légal pour satisfaire le cas uniforme. Nous verrons néanmoins par la suite que la possibilité de produire un code distribué pourra nous aider à étendre la technique aux dépendances non uniformes.

5.4.1 La contraction de tableaux est NP-complète

Dans cette section nous montrons que la maximisation du nombre d'arcs de poids nul par *retiming* se réduit de manière très simple au problème de contraction de tableaux par *retiming*. Tout d'abord, formulons le problème de décision associé à notre problème de contraction.

Problème : Contraction de tableaux (uniforme)

Instance Un graphe de dépendance uniforme $G = (V, E, d)$ et un entier K.

Question Existe-t-il un *retiming* légal r de G tel que

$$|\{u \in V | \forall e = (u, v) \in E, d(e) + r(v) - r(u) = 0\}| \leq K \ ?$$

Notons que dans ce problème de décision nous avons $T = V$, $E_{flot} = E$ et $\forall u \in V, def(u) = u$. Autrement dit chaque sommet correspond à une écriture dans un tableau distinct et tous les arcs sont des dépendances de flot. Par simple réduction, nous montrons que si nous savons résoudre en temps polynomial le cas général présenté précédemment – où l'ensemble de tableaux candidats, l'ensemble de sommets écrivant dans chaque tableau et l'ensemble des dépendances de flot sont donnés – alors nous savons résoudre en temps polynomial le cas simple. Notre preuve prouve donc bien la NP-complétude du cas général. Elle se fait grâce à une construction très simple utilisant les propriétés de conservation du poids d'un chemin par *retiming*.

Théorème 9 *Le problème* Contraction de tableaux (uniforme) *est NP-complet au sens fort.*

Preuve Nous commençons par prouver que le problème appartient à NP, puis nous décrivons notre transformation polynomiale d'une instance de Maximisation des accès locaux en instance de Contraction de tableaux (uniforme), enfin nous montrons l'équivalence entre instances positives.

Le problème est dans NP Comme seuls les arcs de poids nul nous intéressent pour savoir si un tableau est contractable ou non, la preuve est la même que celle de Maximisation des accès locaux : soit une instance positive de notre problème (G, K). Pour tout *retiming* légal r de G solution de notre instance, grâce au lemme 8 nous pouvons construire un *retiming* légal r' de G produisant au moins les mêmes arcs de poids nul que r et dont les valeurs sont bornées par une fonction polynomiale de notre instance. Ce nouveau *retiming* est notre certificat polynomial.

Transformation Le principe de la transformation est de convertir l'objectif d'arc de poids nul en sommet contractable. Soit une instance (G_M, K_M) (où $G_M = (V_M, E_M, d)$) de Maximisation des accès locaux, nous définissons $f(G_M, K_M) = (G, K)$ comme une transformation polynomiale de (G_M, K_M) en instance de Contraction de tableaux (uniforme) de la manière suivante :

- nous partons de $G = G_M$;
- pour chaque arc $e = (u, v) \in E_M$, nous construisons un nouveau sommet x_e. Nous ajoutons également trois arcs e_1, e_2 et e' à E, respectivement de x_e vers u, de x_e vers v et de u à v. Nous définissons $d(e_1) = 0$, $d(e_2) = d(e)$ et $d(e') = d(e) + 1$;
- nous terminons en posant $K = K_M + N$ où N est le nombre de sommets de G n'ayant aucun arc sortant (autrement dit nous considérons un tel sommet comme toujours contractable).

Réduction Soit (G_M, K_M) une instance de Maximisation des accès locaux et sa tranformation par f : $(G, K) = f(G_M, K_M)$. Remarquons tout d'abord que tous les sommets de G n'ayant aucun arc sortant sont systématiquement contractables. Ce qui nous donne de façon fixe N sommets contractables. Par ailleurs, pour tout sommet u de G_M ayant au moins un arc sortant $e = (u, v) \in E_M$, notre transformation ajoute un arc $e' = (u, v)$ à E tel que $d(e') = d(e) + 1$. Donc pour tout *retiming* r de G, $d_r(e') = d(e) + r(v) - r(u) + 1 = d(e) + 1 \neq d(e)$. Autrement dit u ne sera jamais contractable.

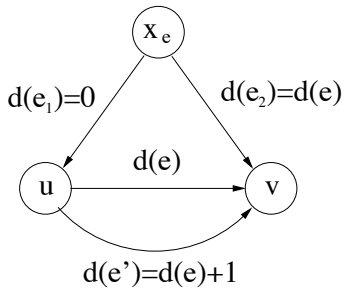


FIG. 5.7 – Transformation d'un arc.

Il ne nous reste plus qu'a considérer les sommets x_e. Pour tout *retiming* r de G_M, nous pouvons étendre r à G en posant $\forall e = (u, v) \in E_M, r(x_e) = r(u)$, il en résulte que $d_r(e_1) = 0+r(x_e)-r(u) = 0$ et $d_r(e_2) = d(e) + r(v) - r(x_e) = d_r(e)$. Comme e_1 et e_2 sont les deux seuls arcs sortant de x_e, il en résulte que x_e est contractable si $d_r(e)$ est nul. Inversement, pour tout *retiming* r de G, pour tout sommet $x_e \in V$ contractable, nous avons $r(u) - r(x_e) = d_r(e_1) = 0$ donc $r(u) = r(x_e)$ et $d(e) + r(v) - r(x_e) = d_r(e_2) = 0$. Finalement nous obtenons $d_r(e) = d(e) + r(v) - r(u) = d(e) + r(v) - r(x_e) = 0$. Donc si x_e est contractable alors $d_r(e)$ est nul. Comme $K = K_M + N$ nous avons l'équivalence entre instances positives et CONTRACTION DE TABLEAUX (UNIFORME) est NP-complet au sens fort. □

Par simple réduction triviale, nous pouvons également en déduire que dans le cas uniforme et distribué le problème est toujours NP-complet (la réduction est triviale puisque, dans le cas uniforme, toute solution distribuée peut se transformer en solution non distribuée). De même, à partir du cas uniforme distribué, nous pouvons déduire que les cas utilisant une représentation plus générale des dépendances (vecteurs de distance, polyèdres, affines, etc.) sont également NP-complets (ici encore la réduction est triviale puisque les représentations plus générales contiennent le cas des dépendances uniformes).

5.4.2 Solution par programmation linéaire

Dans cette section nous proposons une solution par programmation linéaire en nombres entiers au problème de contraction de tableaux par *retiming* et fusion. Nous commençons par traiter le cas uniforme en proposant une transcription directe du programme 5.2 de la page 152 pour la maximisation du nombre d'arcs de poids nul par *retiming*. Nous étendons ensuite ce programme au cas général en permettant la production d'une solution distribuée et l'utilisation de dépendances non uniformes.

Pour le cas uniforme et totalement fusionné, nous pouvons reprendre le programme 5.2 de la page 152 quasiment sans aucune modification. La seule différence est que cette fois le coût est positionné sur les sommets et non sur les arcs. La technique pour différencier les arcs de poids nul des autres reste cependant la même.

Proposition 19 *Soit un graphe de dépendance uniforme $G = (V, E, d)$, T un ensemble de tableaux candidats, def une fonction de T dans l'ensemble des parties de V correspondant aux instructions écrivant dans un tableau de T et $M = \sum_{e \in E} |d(e)|$. Toute solution optimale (r, k) du programme linéaire suivant est telle que r est un* retiming *légal de G maximisant le nombre de tableaux de T*

contractables.

$$\begin{array}{l}\text{Minimiser :}\\ \qquad \sum_{x\in T} k(x)\\ \text{Tel que :}\\ \forall u \in def(x),\\ \forall e = (u,v) \in E_{flot},\\ \qquad Mk(x) \geq d(e) + r(v) - r(u)\\ \forall e = (u,v) \in E,\\ \qquad d(e) + r(v) - r(u) \geq 0\end{array} \tag{5.3}$$

Preuve La preuve est quasiment la même que celle de la proposition 17. Soit une solution optimale (r,k) du programme, à cause des contraintes $d(e) + r(v) - r(u) \geq 0$, r est légal et correspond donc bien à une solution totalement fusionnée. Un tableau n'est pas contractable s'il existe un arc $e = (u,v) \in E_{flot}$ tel que $u \in def(x)$ et $d(e) + r(v) - r(u) > 0$ (non négatif car r est légal). Or $d(e) + r(v) - r(u) > 0 \Rightarrow k(x) \geq 1$ à cause de la contrainte $Mk(x) \geq d(e) + r(v) - r(u)$. Soit K le nombre de tableaux contractables dans G_r, nous avons donc $|T| - K \leq \sum_{x\in T} k(x)$.

Supposons alors que r ne maximise pas le nombre de tableaux contractables, c'est-à-dire qu'il existe un *retiming* légal r_2 de G_{r_2} tel que G_{r_2} ait $K' \geq K + 1$ tableaux contractables. De façon analogue à la la preuve de la proposition 17 (en utilisant le lemme 9), nous montrons qu'il existe r' produisant au moins les mêmes arcs de poids nul que r_2 et tel que $\forall e = (u,v) \in E, d(e) + r'(v) - r'(u) \leq M$. Donc en nous limitant à $k'(x) = 0$ ou $k'(x) = 1$ nous pouvons satisfaire toutes les contraintes de type $Mk'(x) \geq d(e) + r'(v) - r'(u)$. Pour tout tableau $x \in T$ contractable dans G'_r nous avons $\forall u \in def(x), \forall e = (u,v) \in E_{flot}, d(e) + r'(v) - r'(u) = 0$, nous pouvons donc choisir $k'(x) = 0$ pour satisfaire la contrainte $Mk'(x) \geq d(e) + r'(v) - r'(u)$. Comme K' tableaux sont contractables, K' tableaux sont tels que $k'(x) = 0$ et $|T| - K'$ tableaux sont tels que $k'(x) = 1$, donc le coût de (r', k') est $|T| - K' < |T| - K \leq \sum_{x\in T} k(x)$, ce qui contredit l'optimalité de (r,k). □

Cependant, même si d'un point de vue théorique nous pouvions nous restreindre au cas uniforme afin de montrer la complexité du problème, d'un point de vue pratique cette restriction limite nos possibilités. D'une part une solution distribuée peut avoir l'avantage de produire un décalage de moindre importance (et donc un prologue/épilogue plus petit), d'autre part si nous souhaitons étendre la méthode aux dépendances non uniformes, des dépendances interdisant la fusion (quel que soit le *retiming*) peuvent exister entre deux boucles.

5.4.3 Extension aux dépendances non uniformes

La figure 5.8 représente un code comportant une dépendance non uniforme (entre les instructions `e` et `d`). Dans ce cas, la fusion totale est impossible et le *retiming* n'est pas à même d'aligner cette dépendance. Nous pouvons modéliser la distribution de boucles sous forme de contraintes linéaires en reprenant les idées de [78, 9] : pour chaque sommet $u \in V$ de notre graphe, nous ajoutons une variable $\rho(u)$ au programme linéaire représentant le numéro (relatif) de boucle de la séquence dans laquelle se trouve l'instruction associée.

Pour ce qui est des dépendances, afin de traiter le cas général, nous différencierons les arcs en deux types : les arcs que nous appellerons « uniformes » et les arcs que nous appellerons « non uniformes » (selon la nature de la dépendance à laquelle l'arc est associé). Cette distinction vient de notre condition de contractabilité sur un tableau. En effet un tableau n'est contractable que si toutes les dépendances de flot sortant de chaque instruction qui écrit dans ce tableau ont un poids nul. Autrement dit, si une telle dépendance est non uniforme, elle ne peut pas être toujours

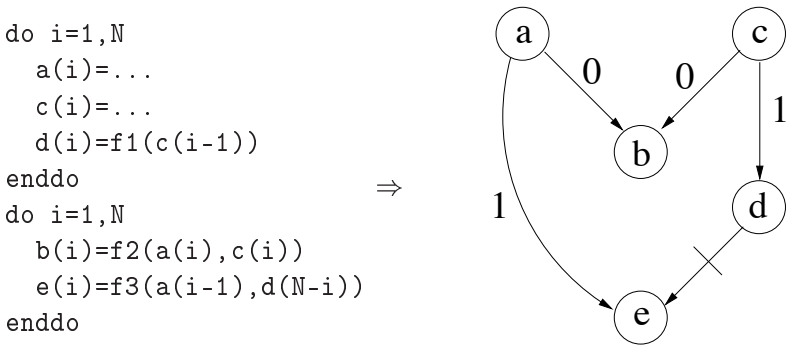


FIG. 5.8 – Exemple de code comportant un arc prévenant la fusion (arc barré du graphe).

nulle et empêche la contraction du tableau (quel que soit le *retiming* utilisé). Par ailleurs, nous différencierons les arcs non uniformes en deux sous types : les arcs prévenant la fusion et les arcs ne prévenant pas la fusion. Cette nouvelle distinction vient de l'ensemble de vecteurs de distance associé à la dépendance. Si toutes les valeurs de cet ensemble sont positives ou nulles, la dépendance n'empêche pas les deux instructions associées de se trouver dans la même boucle. Autrement dit si nous disposons d'une borne inférieure sur les valeurs de cet ensemble, nous pouvons nous en servir et considérer que l'arc n'empêche pas la fusion de manière systématique mais seulement lorsque le *retiming* rend cette borne inférieure négative. Nous dirons donc qu'un arc prévient la fusion lorsque nous ne disposons pas de borne inférieure sur l'ensemble de valeurs que prend la dépendance associée (c'est par exemple le cas des valeurs $-$ ou $*$ pour les vecteurs de direction presentés à la fin du chapitre 4).

Proposition 20 *Soit un graphe de dépendance $G = (V, E, d)$, T un ensemble de tableaux candidats, def une fonction de T dans l'ensemble des parties de V correspondant aux instruction écrivant dans un tableau de T et $M = \max(|V|, \sum_{e\in E} |d(e)| + |V| + \max_{e\in E} |d(e)|)$. Toute solution optimale (r, ρ, k) du programme linéaire suivant est telle que r est un* retiming *de G et ρ définit une distribution légale de G_r maximisant le nombre de tableaux de T contractables.*

$$
\begin{array}{l}
\text{Minimiser :} \\
\qquad \sum_{x\in T} k(x) \\
\text{Tel que :} \\
\forall u \in def(x), \\
\forall e = (u, v) \text{ uniforme} \in E_{flot}, \\
\qquad Mk(x) \geq d(e) + r(v) - r(u) \\
\qquad Mk(x) \geq \rho(v) - \rho(u) \\
\forall u \in def(x), \\
\forall e = (u, v) \text{ non uniforme} \in E_{flot}, \\
\qquad k(x) \geq 1 \\
\forall e = (u, v) \text{ ne prévenant pas la fusion} \in E, \\
\qquad \rho(v) - \rho(u) \geq 0 \\
\qquad M(\rho(v) - \rho(u)) \geq -(d(e) + r(v) - r(u)) \\
\forall e = (u, v) \text{ prévenant la fusion} \in E, \\
\qquad \rho(v) - \rho(u) \geq 1
\end{array}
\tag{5.4}
$$

Preuve La preuve ressemble beaucoup à celle de la proposition 19, sauf que cette fois il faut également s'assurer de la légalité de la distribution.

Soit une solution optimale (r, ρ, k) du programme. Soit P la partition des sommets associée à ρ (où deux sommets u et v appartiennent à la même partition si $\rho(u) = \rho(v)$ et à deux partitions différentes dans le cas contraire). Montrons que P est une distribution légale. Supposons que le sous graphe induit par cette partition contienne un circuit c. Pour tout arc $e = (n_i, n_j) \in c$ (par définition $n_i \neq n_j$), il existe deux sommets $u_i \in n_i$ et $u_j \in n_j$ tels que $(u_i, u_j) \in E$ et $\rho(u_j) - \rho(u_i) \geq 1$ (en effet u_i et u_j n'appartiennent pas au même élément de la partition, donc $\rho(u_j) \neq \rho(u_i)$, et, à cause des contraintes, $\rho(u_j) - \rho(u_i) \geq 0$). En suivant les arcs du circuit, nous obtenons une suite $(u_1, u'_1), (u_2, u'_2), \ldots, (u_n, u'_n)$ d'arcs de E telle que pour $i \in \{1, \ldots, n\}$, $\rho(u'_i) - \rho(u_i) \geq 1$ et pour $i \in \{1, \ldots, n-1\}$, $\rho(u'_i) = \rho(u_{i+1})$ et $\rho(u'_n) = \rho(u_1)$ (en effet, comme nous suivons un circuit de G_P, le puits d'un arc et la source du suivant sont dans le même élément de la partition et ont donc la même valeur ρ). En sommant les inégalités, nous obtenons $\sum_{i=1}^{n} \rho(u'_i) - \sum_{i=1}^{n} \rho(u_i) \geq n$ c'est-à-dire $\rho(u'_n) + \sum_{i=1}^{n-1} \rho(u'_{i+1}) - \sum_{i=1}^{n} \rho(u_i) \geq n$ soit $\rho(u_1) - \rho(u_1) \geq n$, ce qui contredit l'existence d'un circuit dans G_P. Soit $e = (u, v) \in E$ une dépendance, nous avons $\rho(v) - \rho(u) \geq 1$ si la dépendance prévient la fusion, dans le cas contraire $M(\rho(v) - \rho(u)) \geq -(d(e) + r(v) - r(u))$ donc $\rho(v) - \rho(u) = 0 \Rightarrow d(e) + r(v) - r(u) \geq 0$. Donc la partition définie par ρ est une distribution légale.

Cette fois, un tableau n'est pas contractable s'il existe un arc $e = (u, v) \in E_{flot}$ tel que $u \in def(x)$ et $d(e) + r(v) - r(u) > 0$ ou $\rho(v) - \rho(u) \geq 1$ (si le poids est négatif nous aurons de toutes façons $\rho(v) - \rho(u) \geq 1$ car la distribution est légale). Or $d(e) + r(v) - r(u) > 0 \Rightarrow k(x) \geq 1$ à cause de la contrainte $Mk(x) \geq d(e) + r(v) - r(u)$ et $\rho(v) - \rho(u) \geq 1 \Rightarrow k(x) \geq 1$ à cause de la contrainte $Mk(x) \geq \rho(v) - \rho(u)$. Soit K le nombre de tableaux contractables dans G_r, nous avons donc $|T| - K \leq \sum_{x \in T} k(x)$.

Supposons alors que r et ρ ne maximisent pas le nombre de tableaux contractables, il existe un *retiming* r_2 et une distribution légale P' de G_{r_2} produisant $K' \geq K + 1$ tableaux contractables. Comme P' est une distribution légale de G_{r_2}, $(G_{r_2})_{P'} = (P', E_{P'}, (d_r)_{P'})$ est sans circuit. Pour tout sommet $n_i \in P'$, nous définissons $\rho'(n_i) = h(n_i)$ où $h(n_i)$ est la hauteur du sommet n_i dans $(G_{r_2})_{P'}$ (c'est-à-dire la longueur du plus long chemin d'une source quelconque à n_i). Notons que $h(n_i) \leq |V| \leq M$ (au pire il y a $|V|$ éléments dans P' donc une hauteur maximale de $|V|$ dans $(G_{r_2})_{P'}$). Nous étendons ρ' aux sommets de G en posant $\forall n_i \in P', \forall u \in n_i, \rho'(u) = \rho'(n_i)$. Donc pour tout arc (u, v) de E, $\rho'(v) - \rho'(u) = 0$ si u et v appartiennent au même élément de P', $\rho'(v) - \rho'(u) \geq 1$ sinon et $\rho'(v) - \rho'(u) \leq M$ dans tous les cas.

Considérons un élément n_i de P', soit G_{n_i} la restriction de G aux sommets contenus dans n_i. Comme P' est une distribution légale de G_{r_2}, la restriction de r_2 à n_i est un *retiming* légal de G_{n_i}. Donc, d'après le lemme 7 de la page 126, il existe un *retiming* légal r_{n_i} de G_{n_i} produisant exactement les mêmes arcs de poids nul que r_2 et tel que $\forall u \in n_i, 0 \leq r_{n_i}(u) \leq \sum_{e \in E} |d(e)| + |V|$. Nous définissons alors r' de la façon suivante : $\forall n_i \in P', \forall u \in n_i, r'(u) = r_{n_i}(u)$. Comme P' est une partition des sommets, cette définition est unique. Nous avons alors $\forall u \in V, 0 \leq r'(u) \leq \sum_{e \in E} |d(e)| + |V|$, donc $\forall e = (u, v) \in E, -\sum_{e \in E} |d(e)| - |V| - \max_{e \in E} |d(e)| \leq d(e) + r'(v) - r'(u) \leq \sum_{e \in E} |d(e)| + |V| + \max_{e \in E} |d(e)|$ c'est-à-dire $-M \leq d(e) + r'(v) - r'(u) \leq M$. Donc pour tout arc $e = (u, v) \in E$, si $\rho'(v) - \rho'(u) \geq 1$, la contrainte $M(\rho'(v) - \rho'(u)) \geq -(d(e) + r'(v) - r'(u))$ est satisfaite, et si $\rho'(v) - \rho'(u) = 0$ alors u et v sont dans la même partition et $d(e) + r'(v) - r'(u) \geq 0$ (par légalité du r_{n_i} correspondant), donc la contrainte est également satisfaite.

Toutes les contraintes de correction sont maintenant satisfaites, il reste à définir le coût. Pour chaque tableau $x \in T$, si x n'est pas contractable avec r_2 et P' nous définissons $k'(x) = 1$ et toutes les contraintes de coût sont satisfaites. Dans le cas contraire, cela signifie que pour tout $u \in def(x)$, et pour tout $e = (u, v) \in E_{flot}$, u et v sont dans le même élément de P' (dans la même boucle) donc

$\rho'(v) - \rho'(u) = 0$ et $d(e) + r'(v) - r'(u) = 0$ (contractabilité de x et mêmes arcs de poids nul que r_2 à l'intérieur d'un même élément de la partition), donc $k'(x) = 0$ satisfait toutes les contraintes.

Comme K' tableaux sont contractables, K' tableaux sont tels que $k'(x) = 0$ et $|T| - K'$ tableaux sont tels que $k'(x) = 1$, donc le coût de (r', ρ', k') est $|T| - K' < |T| - K \leq \sum_{x \in T} k(x)$ ce qui contredit l'optimalité de (r, ρ, k). □

Remarquons enfin que, dans notre solution au problème de contraction de tableaux, tous les tableaux ont le même coût. Autrement dit, la contraction d'un tableau est toujours aussi profitable quel que soit le tableau. Si nous souhaitons traiter une situation dans laquelle la contraction est plus bénéfique pour certains tableaux que pour d'autres (par exemple lorsque les tableaux ont des tailles différentes), il nous suffit de multiplier $k(x)$ dans la fonction objective par la pénalité induite lorsque le tableau x n'est pas contracté.

5.5 Conclusion

Dans cette section, nous nous sommes intéressés au problème de maximisation du nombre d'arcs de poids nul d'un graphe de dépendance. Après avoir étudié le problème inverse de minimisation au chapitre 2 et après avoir donné un algorithme polynomial pour le résoudre, il nous a semblé intéressant d'un point de vue théorique d'étudier ce problème afin de mieux comprendre le comportement du *retiming*. Une réponse partielle existait déjà dans [23] où il est prouvé que le problème est NP-complet au sens faible pour un *retiming* et un graphe quelconques.

Nous avons prouvé que le problème est en fait NP-complet au sens fort même lorsqu'il est restreint aux *retimings* légaux et aux graphes sans circuits. Nous avons alors proposé une méthode exacte par programmation linéaire pour le résoudre. L'intérêt pour ce problème et les méthodes possibles pour le résoudre est venu de notre volonté de mieux comprendre les mécanismes mis en œuvre par le *retiming*. Toutefois, nous avons illustré comment un problème particulier d'optimisation mémoire peut se ramener à ce problème simplifié de maximisation. Par conséquent, bien qu'au départ motivée par des raisons purement théoriques, la résolution de ce problème peut trouver son utilité dans le cadre de problématiques plus complexes. En outre, la version simplifiée du problème que nous avons étudiée donne une bonne idée sur la complexité générale de ses variantes plus élaborées.

Conclusion

Dans cette thèse nous avons étudié de multiples aspects de la transformation de programmes par décalage d'instructions. Dans chacun des contextes auxquels nous nous sommes intéressés, nous avons souhaité caractériser le décalage d'instructions en termes de complexité et de pouvoir d'expression. Dans tous les cas, nous avons fourni des moyens pratiques de mise en œuvre de cette transformation. Le plus surprenant dans l'ensemble de nos recherches est l'unité qui finit par se dégager de tous ces problèmes de décalage d'instructions. Au fil des résultats, positifs ou négatifs, les grandes lignes des mécanismes sous-jacents à cette transformation se dessinent peu à peu.

Pour une simple boucle, nous nous sommes tout d'abord penchés sur la problématique du pipeline logiciel, c'est-à-dire de la recherche d'un ordonnancement cyclique pour les opérations d'une boucle. Comme nous l'avons rappelé en section 2.3, tout ordonnancement cyclique peut s'exprimer comme la combinaison d'un décalage et d'une compaction. Nous avons alors cherché à comprendre quelle influence le décalage d'instruction avait sur la compaction, et comment il était possible de bien choisir le décalage afin de bien compacter la boucle. Suite à cette réflexion, nous avons proposé une amélioration à une technique de pipeline logiciel décomposé fondée sur le décalage d'instructions, l'algorithme de Calland, Darte et Robert. Notre amélioration consiste en un algorithme de flot polynomial pour la minimisation du nombre de contraintes pour la compaction. Exprimée autrement, cette optimisation cherche à déterminer le décalage le plus à même de produire un maximum de parallélisme au niveau des instructions. Grâce à l'outil PASTAGA, que nous avons développé pour l'occasion, nous avons pu valider expérimentalement cette technique. Nos objectifs étaient de montrer d'une part le bien-fondé de notre optimisation, et d'autre part l'efficacité absolue des algorithmes de pipeline logiciel décomposé. À la lumière de nos résultats il semblerait que, bien que le pipeline logiciel soit un problème difficile, un algorithme polynomial fondé sur le décalage peut produire un résultat suffisamment bon pour ne pas nécessiter l'emploi d'une méthode plus complexe. Cependant, il apparaît que certaines situations restent difficiles pour l'approche décomposée, et l'étude d'autres optimisations permettrait de pousser encore plus loin notre compréhension du problème.

Toujours pour une simple boucle, nous avons ensuite étudié le problème du réordonnancement par étage, qui complète de façon naturelle celui du pipeline logiciel. Nous avons pu classer ce problème, dont la complexité restait inconnue, en montrant qu'il est NP-complet au sens fort. Malgré ce résultat négatif, nous avons proposé une formulation exacte du problème par programmation linéaire en nombre entiers. Bien que de complexité *a priori* non polynomiale, la formulation de notre programme est de taille suffisamment réduite pour envisager son utilisation. Néanmoins, afin de pouvoir mettre en œuvre le réordonnancement par étages dans des programmes de grande taille, nous avons également conçu une méthode d'approximation polynomiale du résultat dont la performance est garantie. Nous avons vérifié expérimentalement l'efficacité de notre approche, et nous avons pu constater que lorsque la taille du problème devient trop grande pour l'approche exacte, l'approximation polynomiale est toujours extrêmement efficace. Il nous reste à comprendre pourquoi

l'approximation par excès apparait meilleure que l'approximation par défaut lors des expérimentations. Peut-être ceci cache-t-il une possibilité d'amélioration de la borne fournie par la garantie.

Enfin, pour en terminer avec le cas d'une simple boucle, nous avons étudié le problème de l'« alignement » des dépendances sous deux points de vue différents. Le problème de l'alignement total est un problème bien connu. Son résultat est de rendre la boucle parallèle par simple décalage, lorsqu'un tel décalage existe. Ce problème est soluble polynomialement, mais les contraintes à satisfaire sont très fortes, et une solution existe rarement. À l'inverse, lorsque l'alignement total des dépendances est impossible, nous pouvons envisager le problème de la maximisation des dépendances alignées (ou maximisation du nombre d'arcs de poids nul). Ce problème est exactement le problème inverse de celui étudié dans le cadre du pipeline logiciel. Nous avons alors montré que ce problème est NP-complet au sens fort et nous avons proposé une méthode pour le résoudre de façon exacte par programmation linéaire en nombres entiers. Bien que la forme de ce problème soit très générale, nous avons montré qu'un objectif d'optimisation précis – la contraction de tableaux – peut s'exprimer comme un problème de maximisation du nombre d'arcs de poids nul. Il reste à déterminer si les différentes variantes de problèmes d'optimisation mémoire sont réellement plus difficiles ou si certaines ont un comportement différent.

Nous nous sommes également penchés sur l'extension des problèmes de décalage d'instructions au cas multi-dimensionnel, c'est-à-dire au cas de nids de boucles imbriquées. Nous avons pu constater que dans ce contexte, les problématiques simples ne change pas : l'algorithme d'alignement de toutes les boucles (algorithme 11, page 120) ou celui de décalage sous contrainte (algorithme 2, page 22) restent essentiellement les mêmes dans le cas multi-dimensionnel[3], et le problème de parallélisme au niveau des instructions se ramène au cas d'une simple boucle (théorème 5, page 124). En revanche, le cas multi-dimensionnel offre la possibilité de combiner plusieurs techniques de décalage différentes sur chacune des boucles du nid. Dans cette optique, nous avons étudié le problème de décalage sur la boucle externe afin de permettre à la boucle interne d'être parallélisée par un second décalage. Nous avons montré que ce problème est NP-complet au sens fort et nous avons proposé une méthode pour le résoudre de façon exacte par la satisfaction d'un ensemble de contraintes linéaires. Nous avons également proposé des méthodes d'identification de cas particuliers solubles en temps polynomial, et une heuristique permettant de résoudre le problème général de façon partielle. Pour ces derniers résultats, nos premiers tests expérimentaux n'ont pas illustré de façon significative les spécificités de l'approche. Une étude expérimentale poussée permettrait de déterminer dans quelle mesure l'approche de la parallélisation par décalage est profitable en pratique et quelle proportion des codes réels se laissent paralléliser par l'un de nos algorithmes polynomiaux.

À la suite de tous ces résultats, nous pouvons alors formuler un ensemble de remarques générales sur le décalage d'instructions. Du point de vue algorithmique, il s'exprime comme un potentiel dans un graphe, d'où son lien très fort avec les algorithmes de plus court chemin et de flot. Dans le cadre de la transformation de code, par exemple, les contraintes de correction ou de parallélisme sur le résultat se traduisent de façon naturelle en inégalités triangulaires sur les valeurs de décalage de deux instructions. De façon analogue, lors de la manipulation du graphe en vue de la réalisation d'un objectif précis, c'est la structure du graphe qui va limiter le pouvoir d'expression du décalage d'instructions. Ces contraintes structurelles vont se traduire sous la forme d'une notion duale : celle de flot dans le graphe. Le plus extraordinaire est la richesse et la variété des problèmes qui s'expriment simplement par une transformation de la structure du graphe – par l'ajout d'arcs ou de sommets supplémentaires – et la satisfaction de simples contraintes d'inégalités triangulaires ou de flot. Du point de vue de leur résolution, les problèmes de décalage d'instructions sont soit

[3]Excepté pour les contraintes de correction qui peuvent entraîner un décalage supplémentaire dans les dimensions les plus internes.

simples, et dans ce cas ils s'expriment comme des problèmes de flot ou de plus court chemin, soit complexes et NP-complets au sens fort. Notons que, bien qu'un décalage soit une valeur entière, les problèmes de décalage ne sont pas intrinsèquement des problèmes de nombres. En effet, tous nos résultats de NP-complétude sont au sens fort, autrement dit la difficulté du décalage d'instructions est de nature combinatoire. Elle provient soit de la structure du graphe (l'alignement total d'une forêt est simple, mais la maximisation du nombre d'arcs de poids nuls d'un graphe quelconque est contrainte par ses cycles), soit de la complexité de l'objectif (pour le réordonnancement par niveau, le coût en fonction du décalage est imprévisible), soit enfin de la combinaison de deux problèmes de décalage antagonistes (pour la parallélisation de la boucle interne, aussi bien le décalage externe seul que le décalage interne seul sont polynomiaux ; c'est la combinaison des deux qui rend le problème NP-complet).

En conclusion, malgré sa simplicité apparente, le décalage d'instructions peut réserver quelques surprises. Bien que largement utilisé dans le cadre de la transformation de programmes, son importance, sa complexité et ses possibilités étaient jusqu'alors peu comprises. Outre le fait d'acquérir une meilleure compréhension de cette technique, nos travaux avait également pour but de convaincre le lecteur de la richesse des problématiques de décalage d'instructions. Pour finir, nous avons regroupé les quelques problèmes de transformation de programme par décalage connus dans la table 5.1.

Problème	Domaine	Complexité	Commentaire
Fusion séquentielle avec décalage (minimisation du nombre de boucles)	fusion de boucles	$O(\|V\|\|E\|)$	un *retiming* légal suffit : algorithme 2, voir aussi [69]
Fusion parallèle avec décalage (minimisation du nombre de boucles)	fusion de boucles	NP-complet	voir [9]
Minimisation du chemin critique	pipeline logiciel	$O(\|V\|\|E\| \lg \|V\|)$	algorithme de Leiserson et Saxe [70], étendu en section 2.3.4
Minimisation du nombre d'arcs de poids nul – corps complètement parallèle – avec contrainte de chemin critique	pipeline logiciel	$O(\|E\|^2)$ $O(\|V\|\|E\|)$ $O(\|E\|(\|E\|+\|V\|^2))$	programme linéaire dans [10], algorithme de flot en section 2.4 cas particulier, section 2.4.1 extension, section 2.4.3
Réordonnancement par étages – programme linéaire – par approximation du coût (deux variantes)	optimisation mémoire	NP-complet selon l'algorithme de flot utilisé	section 3.4.1, NP-complet avec : – deux arcs entrants par sommet – graphes sans circuits solution exacte, section 3.4.3 solution à $\|V\|$ près, section 3.4.5
Parallélisation de boucle (alignement)	parallélisation de boucles	$O(\|V\|+\|E\|)$	section 4.3, voir aussi [89]
Parallélisation d'une boucle interne d'un nid – système de contraintes – par décalage interne – par décalage externe – par décomposition en arbre couvrant	parallélisation de boucles	NP-complet $O(\|V\|+\|E\|)$ $O(\|V\|\|E\|)$ $O(\|V\|\|E\|)$	section 4.4.2, NP-complet en dimension 2 solution exacte, section 4.4.4 cas particulier, section 4.5.1 cas particulier, section 4.5.2 heuristique, section 4.5.3
Maximisation du nombre d'arcs de poids nul – programme linéaire	optimisation mémoire	NP-complet	section 5.2, NP-complet avec : – décalage légal ou non – graphes sans circuits solution exacte, section 5.3
Contraction de tableaux par décalage – programme linéaire	optimisation mémoire	NP-complet	ou encore privatisation de tableaux, section 5.4.1 solution exacte, section 5.4.2 cas non uniforme, section 5.4.3

TAB. 5.1 – Quelques problèmes de décalage d'instructions.

Bibliographie

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *COMPILATEURS, Principes, techniques et outils*. Addison-Wesley Europe, 1991.

[2] Alexander Aiken and Alexandru Nicolau. Perfect pipelining : A new loop optimization technique. In *1988 European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 221–235. Springer-Verlag, 1988.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3) :367–432, September 1995.

[4] John Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, Germany, January 1987.

[5] Tsing-Fa Lee Allen, C.-H Wu Wei-Jeng Chen, Wei-Kai Cheng, and Youn-Long Lin. On the relationship between sequential logic retiming and loop folding. In *Proceedings of the SASIMI'93*, pages 384–393, Nara, Japan, October 1993.

[6] AMD. *AMD Athlon Processor x86 Code Optimization Guide*, August 2001. Electronic form at `http://www.amd.com/products/cpg/athlon/techdocs/index.html`.

[7] Denis Barthou. *Analyse du flot de données pour tableaux en présence de cointraintes non-affines*. PhD thesis, Université de Versailles, 1998.

[8] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15 :757–762, October 1966.

[9] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms : From parallelism extraction to code generation. *Journal of Parallel Computing*, 24(3), 1998. Special issue on Languages and Compilers for Parallel Computers.

[10] Pierre-Yves Calland, Alain Darte, and Yves Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1) :24–35, January 1998.

[11] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes, Rennes, France, 1993. numéro 957.

[12] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[13] L.-F. Chao, A. LaPaugh, and E. Sha. Rotation scheduling : A loop pipelining algorithm. In *30th ACM/IEEE Design Automation Conference*, pages 566–572, 1993.

[14] J.C. Chung. Optimal loop parallelization based on a retiming technique. In *International Conference on Parallel Processing*, 1992.

[15] Albert Cohen. *Analyse et transformation de programmes : du modèle polyhédrique aux langages formels*. PhD thesis, Université de Versailles, 1999.

[16] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3) :421–436, September 1995.

[17] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy Array Dataflow Analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[18] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[19] Cray. Cray inc. supercomputers. Electronic document `http://www.cray.com`.

[20] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. A constructive solution to the juggling problem in systolic array synthesis. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 815–821, Cancun, Mexico, May 2000.

[21] Alain Darte. On the complexity of loop fusion. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 149–157, Newport Beach, CA, October 1999.

[22] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9) :1175–1193, 2000.

[23] Alain Darte and Yves Robert. A graph-theoretic approach to the alignment problem. Technical Report 93-20, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, July 1993.

[24] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8) :814–822, 1994.

[25] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000. ISBN 0-8176-4149-1.

[26] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4) :379–392, 1997.

[27] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6) :447–497, December 1997.

[28] Dominique de Werra. *Eléments de programmation linéaire avec applications aux graphes*. Presses polytechniques romandes, 1 edition, 1990. ISBN 2-88074-176-9.

[29] J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. *Journal of Supercomputing*, 7(1), January 1993.

[30] Claude G. Diderich and Marc Gengler. The alignment problem in a linear algebra framework. In *Proceedings of the Hawaï International Conference on System Sciences (HICSS-30), Software Technology Track*, pages 586–595, Wailea, HI, 1997. IEEE Computer Society Press.

[31] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. Technical Report CRPC-TR99808-S, Rice University, Center for Research on Parallel Computation, May 1999.

[32] Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7) :24–32, July 1998.

[33] A. Eichenberger and E. S. Davidson. Stage scheduling : A technique to reduce the register requirements of a modulo schedule. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 338–349, 1995.

[34] A. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 75–84, San Jose, CA, 1994.

[35] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 31–40, Barcelona, Spain, 1995.

[36] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimizing register requirements of a modulo schedule via optimum stage scheduling. *International Journal of Parallel Programming*, 24(2) :103–132, 1996.

[37] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph : a new model for loop cyclic register allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the PACT'95 Conference on Parallel Architectures and Compilation Techniques*. ACM Press, June 1995.

[38] M. Gondran et M. Minoux. *Graphes et algorithmes*. Eyrolles, 1985.

[39] Paul Feautrier. Array expansion. In *ACM International Conference on Supercomputing*, pages 429–441, 1988.

[40] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1) :23–51, 1991.

[41] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I : One-dimensional time. *International Journal of Parallel Programming*, 21(5) :313–348, October 1992.

[42] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II : Multi-dimensional time. *International Journal of Parallel Programming*, 21(6) :389–420, December 1992.

[43] Paul Feautrier. Towards automatic distribution. *Parallel Processing Letters*, 4(3) :233–244, 1994.

[44] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires : mode d'emploi du logiciel PIP. Technical Report 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), January 1990.

[45] Guang R. Gao, Russ Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. Technical Report 41, McGill University, School of Computer Science, March 1992.

[46] Guang R. Gao and Vivek Sarkar. Optimization of array accesses by collective loop transformations. Technical Report 22, McGill University, School of Computer Science, December 1990.

[47] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[48] F. Gasperoni and U. Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Processing Letters*, 4(4) :391–403, 1994.

[49] F. Gasperoni and U. Schwiegelshohn. Transforming cyclic scheduling problems into acyclic ones. In P. Chrétienne, E. G. Coffman, Jr., J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and Its Applications*, pages 241–258. John Wiley & Sons, 1995.

[50] Johan De Gelas. Floating-point compiler performance analysis. Electronic document `http://www.aceshardware.com/Spades/read.php?article_id=40000189`.

[51] Johan De Gelas. Pentium 4 architecture in depth, part 1. Electronic document `http://www.aceshardware.com/Spades/read.php?article_id=20000190`.

[52] Johan De Gelas. Pentium 4 architecture in depth, part 2. Electronic document `http://www.aceshardware.com/Spades/read.php?article_id=25000191`.

[53] R. Govindarajan, E.R. Altman, and G.R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *27th Symposium on Microarchitectures*, 1994.

[54] C. Hanen and A. Munier. Cyclic scheduling on parallel processors : An overview. In P. Chrétienne, E. G. Coffman, Jr., J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and Its Applications*. John Wiley & Sons, 1995.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach (2nd Edition)*. Morgan-Kaufmann, 1996.

[56] Ping Hu. Ordonnancement modulo par recouvrement. In *10ème rencontres du parallélisme (RenPar'10)*, Strasbourg, France, June 1998.

[57] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Conference on Programming Language Design and Implementation (PLDI'93)*, pages 258–267. ACM, 1993.

[58] Intel. *Intel Architecture Optimization*, 1999. Electronic form at `http://developer.intel.com/design/PentiumIII/manuals/`.

[59] Intel. *Intel IA-64 Architecture Software Developper's Manual*, 2000. Electronic form at `http://developer.intel.com/design/itanium/manuals/index.htm`.

[60] Intel. *Intel Pentium 4 Processor Optimization*, 2001. Electronic form at `http://developer.intel.com/design/Pentium4/manuals/`.

[61] Suneel Jain. Circular scheduling. In *Conference on Programming Language Design and Implementation (PLDI'91)*, pages 219–228. ACM, 1991.

[62] W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. In *Proceedings of the tenth ACM International Conference on Supercomputing*. ACM Press, 1996.

[63] Wayne Kelly and Bill Pugh. Selecting affine mappings based on performance estimation. *Parallel Processing Letters*, 4(3) :205–219, September 1994.

[64] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[65] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing'90*, August 1990.

[66] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[67] Monica S. Lam. Software pipelining : An effective scheduling technique for VLIW machines. In *SIGPLAN'88 Conference on Programming Language, Design and Implementation*, pages 318–328, Atlanta, GA, 1988. ACM Press.

[68] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2) :83–93, February 1974.

[69] C. Lang, N. Passos, and E. Sha. Polynomial-time nested loop fusion with full parallelism. In *International Conference on Parallel Processing*, volume 3, pages 9–16, Bloomingdale, IL, 1996.

[70] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1) :5–35, 1991.

[71] Sylvain Lelait, Guang R. Gao, and Christine Eisenbeis. A new fast algorithm for optimal register allocation in modulo scheduled loops. Technical Report 3337, Institut national de recherche en informatique et automatique, January 1998.

[72] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

[73] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, 1992.

[74] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1997.

[75] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Principles Practice of Parallel Programming*, pages 103–112, 2001.

[76] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling : A lifetime-sensitive approach. In *Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, Boston, MA, 1996. IEEE Computer Society Press.

[77] Kathryn S. McKinley and Ken Kennedy. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *The Sixth Annual Languages and Compiler for Parallelism Workshop*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer-Verlag, 1993.

[78] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

[79] Darek Mihocka. Pentium 4 : in depth. Electronic document `http://www.emulators.com/pentium4.htm`.

[80] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *25th Annual International Symposium on Microarchitecture*, pages 55–71, 1992.

[81] Myricom, Inc. LANai 3.0 instruction set. Electronic document `http://www.myricom.com/scs/L3/doc/inst_toc.html`.

[82] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In 20^{th} *ACM Symposium on Principles of Programming Languages*, 1993.

[83] Kunio Okuda. Cycle shrinking by dependence reduction. In *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 398–401. Springer-Verlag, 1996.

[84] Thomas Pabst. Important pentium 4 evaluation update. Electronic document `http://www4.tomshardware.com/cpu/00q4/001122/index.html`.

[85] Thomas Pabst. Intel's new pentium 4 processor. Electronic document `http://www4.tomshardware.com/cpu/00q4/001120/index.html`.

[86] Thomas Pabst. Painting a new picture of pentium 4 - tweaked mpeg4 encoding. Electronic document `http://www4.tomshardware.com/cpu/00q4/001125/index.html`.

[87] Thomas Pabst. Pentium 4 vs. athlon - final recount. Electronic document `http://www4.tomshardware.com/cpu/00q4/001206/index.html`.

[88] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12) :1184–1201, December 1986.

[89] J. K. Peir. *Program Partitioning and Synchronization on Multiprocessor Systems*. PhD thesis, University of Illinois at Urbana-Champaign, March 1986. Report UIUC-DCS-R-86-1259.

[90] J. K. Peir and R. Cytron. Minimum distance : A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8) :1203–1211, August 1989.

[91] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5), October 2000.

[92] M. Rajagopalan and V. H. Allan. Specification of software pipelining using petri nets. *International Journal of Parallel Programming*, 22(3) :273–301, 1994.

[93] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1) :3–64, 1996.

[94] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the Fourteenth Annual Workshop of Microprogramming*, pages 183–198, October 1981.

[95] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *SIGPLAN'92 Conference on Programming Language, Design and Implementation*, pages 283–299, Atlanta, GA, 1992. ACM Press.

[96] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 158–169, Portland, Oregon, December 1992. IEEE Computer Society Press.

[97] Antoine Sawaya. *Pipeline logiciel : découplage et contraintes de registres*. PhD thesis, Université de Versailles, France, 1997.

[98] Robert Schreiber, Shail Aditya, Bob R. Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical Report HPL-2000-31, Hewlett-Packard Labs, 2000.

[99] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[100] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On optimal parallelization of arbitrary loops. *Journal of Parallel and Distributed Computing*, 11 :130–134, 1991.

[101] Jon Stokes. The pentium 4 and the g4e : an architectural comparison. Electronic document `http://www.arstechnica.com/cpu/01q2/p4andg4e/p4andg4e-1.html`.

[102] Jon Stokes. Sound and vision : a technical overview of the emotion engine. Electronic document `http://www.arstechnica.com/reviews/1q00/playstation2/ee-1.html`.

[103] Peng Tu and David A. Padua. Automatic array privatization. In *1993 Workshop on Languages and Compilers for Parallel Computing*, volume 768, pages 500–521, Portland, Ore., 1993. Berlin : Springer Verlag.

[104] Frédéric Vivien. *Détection de parallélisme dans les boucles imbriquées*. PhD thesis, École normale supérieure de Lyon, France, December 1997.

[105] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining. *International Journal of Parallel Programming*, 22(3) :351–373, 1994.

[106] N. J. Warter, G. E. Haab, and J. W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, Portland, Oregon, December 1992. IEEE Computer Society Press.

[107] N. J. Warter, S. A. Mahlke, W. M. W. Hwu, and B. R. Rau. Reverse if-conversion. In *PLDI*, pages 290–299, New York, June 1993. ACM.

[108] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4) :452–471, October 1991.

[109] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[110] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

Publications

– **Journaux :**

Darte (Alain) et Huard (Guillaume). – Loop shifting for loop compaction. *International Journal of Parallel Programming*, 28(5), 2000.

– **Conférences internationales :**

Darte (Alain) et Huard (Guillaume). – Complexity of multi-dimensional loop alignment. *Symposium on Theoretical Aspects of Computer Science Proceedings.* Springer Verlag, 2002.

Darte (Alain) et Huard (Guillaume). – Loop shifting for loop compaction. *Twelfth International Workshop on Languages and Compilers for Parallel Computing Proceedings*, éd. par Carter et Ferrante. – Springer Verlag, 1999.

Fraboulet (Antoine), Huard (Guillaume) et Mignotte (Anne). – Loop alignment for memory access optimization. *Twelfth International Symposium on System Synthesis Proceedings*, IEEE Computer Society Press. – 1999.

– **Conférences nationales :**

Huard (Guillaume). – Parallélisation de boucles par décalage d'instructions. *$13^{ièmes}$ Rencontres Francophones du Parallélisme des Architectures et des Systèmes*, éd. par Cérin et Utard. – RenPar/CFSE/SympA, 2001.

– **Rapports de recherche :**

Darte (Alain) et Huard (Guillaume). – *Loop shifting for loop parallelization.* – Rapport technique N° RR2000-22, ENS-Lyon, France, LIP, mai 2000.

Darte (Alain) et Huard (Guillaume). – *Loop shifting for loop compaction.* – Rapport technique N° RR1999-29, ENS-Lyon, France, LIP, mai 1999.

Fraboulet (Antoine), Huard (Guillaume) et Mignotte (Anne). – *Loop alignment for memory access optimization.* – Rapport technique N° RR1999-26, ENS-Lyon, France, LIP, 1999.

Huard (Guillaume). – *Retiming et parallélisation automatique.* – Rapport technique N° RR1998-33, ENS-Lyon, France, LIP, juillet 1998. Rapport de DEA, sous la direction d'Alain Darte.

Résumé

L'évolution constante des processeurs vers des architectures proposant des capacités superscalaires, de parallélisme au niveau des instructions, de prédiction, de spéculation et la multiplication des niveaux de hiérarchie mémoire donnent de plus en plus d'importance au travail du compilateur. Dans cette thèse, nous nous intéressons aux transformations du programme source destinées à l'optimisation dans la chaîne de compilation, et plus particulièrement à une transformation appelée décalage d'instructions. Cette transformation sert de base au pipeline logiciel, elle a une influence sur le parallélisme au niveau des instructions et l'utilisation des registres. Elle intervient également comme composante des techniques de parallélisation de boucles par ordonnancement affine. Nous avons voulu mieux comprendre les perspectives offertes par le décalage d'instructions, savoir quels objectifs il permettait d'atteindre mais aussi savoir quels problèmes de décalage restaient difficiles. Pour cela nous avons étudié le décalage d'instructions dans plusieurs contextes plus ou moins proches, et apporté des contributions à chacun d'entre eux.

Dans le cadre du pipeline logiciel, nous proposons un algorithme polynomial pour déterminer le décalage le plus à même de produire un maximum de parallélisme au niveau des instructions, et une étude expérimentale de l'efficacité absolue de la technique à l'aide de l'outil logiciel que nous avons réalisé dans ce but : PASTAGA (pour Plate-forme d'Analyse Statistique et de Tests d'Algorithmes sur Graphes Aléatoires). Dans le cadre de l'utilisation des registres (*stage scheduling*), de la parallélisation de boucle et de la localité, nous apportons des réponses aux problèmes de décalage d'instructions associés : complexité, solutions exactes, approximations.

Mots clés : Compilation, transformations de programme, parallélisme, décalage d'instructions, *retiming*, pipeline logiciel, ordonnancement, parallélisation automatique.

Abstract

The constant evolution of processors architectures, with superscalar, instruction-level parallelism, prediction and speculation capabilities and the multiple number of levels in the memory hierarchy give an increasing importance to the work of the compiler. In this thesis we deal with source program transformations, intended to optimization within the compilation process, and especially with a transformation known as loop shifting. This transformation is used as a basis for software pipelining, it has an incidence on the instruction level parallelism and registers usage. It is also involved as a component of loop parallelization techniques based on affine schedules. In this thesis we have tried to reach a better understanding of the possibilities that loop shifting has to offer, to know which goals it is able to score, and which problems remain hard. To this intent, we have studied loop shifting within a variety of contexts, more or less close to each other, and we provide a contribution for each of them.

In the context of software pipelining, we give a polynomial algorithm to find the loop shifting leading to as much instruction-level parallelism as possible, and we perform an experimental study of its absolute efficiency, with the help of PASTAGA (french translation of "Platform for statistical analysis and algorithms testing on random graphs"), a tool we developed for this purpose. In the contexts of register usage (stage scheduling), loop parallelization and locality, we give answers in each case to loop shifting problems : complexity, exact solutions and heuristics.

Keywords : Compilation, program transformations, parallelism, loop shifting, retiming, software pipelining, scheduling, automatic parallelization.