Université de Montréal

Estimation des performances du système *PULSE V1* pour des applications de nature itérative

par

Noureddine Chabini

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

Juin, 1998

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé

Estimation des performances du système *PULSE V1* pour des applications de nature itérative

présenté par

Noureddine Chabini

a été évaluée par un jury composé des personnes suivantes:

Marc Feeley	président du jury
El Mostapha Aboulhamid	directeur de recherche
Yvon Savaria	co-directeur de recherche
Xiaoyu Song	membre du jury

Mémoire accepté le: 05.11.1998

Sommaire

Dans ce mémoire, nous nous intéressons à l'estimation des performances de systèmes informatiques parallèles de type "Single Instruction Stream over Multiple Data Streams" (SIMD). La motivation principale de ce travail est le besoin d'estimation des performances d'un ordinateur parallèle de type SIMD en cours de conception appelé *PULSE V1* destiné pour des applications réelles. Ce dernier se compose de quatre processeurs parallèles identiques de type "Very Long Instruction Word" (VLIW) ayant des ressources de type "pipeline". L'objectif de ce mémoire est de développer un outil informatique pour estimer les performances de systèmes parallèles de type SIMD et de l'appliquer à l'estimation des performances de l'ordinateur *PULSE V1*.

La programmation d'un système parallèle de type SIMD se caractérise par l'exécution d'un seul programme sur les processeurs de ce système sous le contrôle d'une seule unité de commande appelée contrôleur; tous les processeurs reçoivent la même instruction diffusée par le contrôleur, mais opèrent sur des données provenant de flux de données distincts. Par conséquent, en supposant que les processeurs de ce système sont identiques et que les communications inter-processeurs sont uniformes, l'estimation de ses performances revient à estimer les performances d'un de ses processeurs. Notons que ces derniers peuvent être des processeurs parallèles de type VLIW ayant des ressources de type "pipeline".

Nous avons développé un outil informatique pour estimer les performances de systèmes parallèles de type SIMD. Cet outil est une implémentation d'une méthode statique d'estimation de performance. Cette méthode se base sur des algorithmes d'ordonnancement, d'allocation, et de calcul de plus court et de plus long chemins. Ensemble ces algorithmes permettent de déterminer des bornes inférieure et supérieure sur les performances.

Les algorithmes de cette méthode ont besoin d'un graphe de contrôle et de flot de données (GCFD) qui est une représentation intermédiaire d'un programme donné. Si le GCFD était généré pour une exécution séquentielle, des dépendances de données supplémentaires doivent être ajoutées dans celui-ci puisque du parallélisme sera cherché. En plus, les informations suivantes

doivent être présentes dans le GCFD quand ce dernier correspond à un traitement itératif. Ces informations sont le début et la fin de chaque boucle, ainsi que ses nombres minimal et maximal d'itérations. Il existe des compilateurs, tel que le compilateur *LCC*, qui génèrent un GCFD. Cependant, ils ne répondent pas aux besoins de ces algorithmes. Pour cela, nous avons apporté des modifications et des extensions au compilateur *LCC*. Afin de visualiser graphiquement le GCFD, nous avons implémenté une interface permettant de communiquer le compilateur *LCC* modifié avec un outil de visualisation graphique d'un graphe.

En général, le partitionnement d'un programme décrit dans un langage séquentiel en des sous-programmes dont l'exécution ne requiert pas de communications interprocesseurs peut ne pas être immédiat. Ceci découle du fait que le parallélisme dans ce programme peut être caché. Afin de faire apparaître ce parallélisme, des transformations de celui-ci sont requises. Pour cela, nous présentons un processus et des techniques de parallélisation automatique d'un programme. Ces techniques sont destinées à des traitements itératifs.

Nous avons utilisé l'outil développé pour estimer les performances de l'ordinateur *PULSE VI* en exécutant certaines applications du domaine de traitement d'images. Ces applications correspondent au calcul d'un ensemble d'éléments de sortie des filtres suivants: réponse impulsionnelle fini unidimensionnelle, réponse impulsionnelle fini bidimensionnelle, et réponse impulsionnelle infini unidimensionnelle. Nous avons obtenu des résultats quasi-optimaux.

Table des matières

Liste des tableaux

Liste des figures

Remerciements

Qu'il me soit permis de remercier vivement mon directeur de recherche Monsieur le Professeur El Mostapha Aboulhamid. Son aide financière, ses encouragements soutenus, ses conseils précieux et la confiance qu'il m'a accordée m'ont beaucoup aidé tout au cours de mes études supérieures.

Que mon co-directeur de recherche Monsieur le Professeur Yvon Savaria trouve ici l'expression de toute ma reconnaissance. Son aide financière, ses encouragements continus, ses conseils judicieux et la confiance qu'il m'a témoignée m'ont beaucoup aidé à la réalisation de ce travail.

Je tiens à exprimer mes profonds remerciements aux membres du jury de ce mémoire, Messieurs le Professeur Marc Feeley, le Professeur El Mostapha Aboulhamid, le Professeur Yvon Savaria, et le Professeur Xiaoyu Song, pour l'évaluation de ce travail.

Je remercie le Département d'informatique et de recherche opérationnelle qui a contribué à ma formation. Que son corps professoral trouve ici l'expression de toute ma gratitude.

Je tiens à remercier l'Université de Montréal de m'avoir octroyé une bourse pour continuer mes études doctorales en sciences informatiques au sein de son Département d'informatique et de recherche opérationnelle.

Je remercie toutes les personnes du laboratoire LASSO et du groupe PULSE. Vous avez su créer une ambiance de travail exemplaire.

Je veux aussi remercier le Professeur Bernard Gendron pour les nombreuses et agréables discussions que nous avons eues.

Je dois beaucoup a une personne qui m'est très chère: mon frère Ismaïl. Depuis mon jeune âge, Ismaïl m'a toujours servi comme un rare et précieux "role model" dans la poursuite de mes études. Ismaïl a aussi contribué, très généreusement, aux financements de mes études. Malgré la charge, si énorme, d'un Professeur du MIT, Ismaïl a toujours su trouver le temps pour me visiter, m'aider et me conseiller. Ismaïl, merci infiniment.

Finalement, tout mon respect va à ma famille pour ses nombreux sacrifices.

À la mémoire de mon père

À ma mère

À mon frère Omar

À mes soeurs, à mes frères

Introduction

1- Système et performance

Un système est un ensemble de composants. Ces composants travaillent ensemble pour réaliser une certaine tâche. La réalisation de cette tâche représente la fonctionnalité de ce système. Pour illustrer le concept de système, prenons l'exemple d'un ordinateur. Dans sa forme la plus simple, les composants d'un ordinateur sont: l'unité de contrôle, l'unité de traitement (ou processeur), la mémoire et les périphériques d'entrées-sorties. Une tâche à réaliser par l'ensemble de ces composants est le traitement de l'information. Dans ce mémoire, nous nous intéressons à des systèmes informatiques.

Car chaque système a ses propres performances, il n'y a pas une définition absolue pour le concept de performance. Cependant, nous illustrons ce concept pour un ordinateur, notre système préféré, en présentant certains paramètres caractérisant ses performances. Deux exemples de ces paramètres sont le temps de réponse et le débit. Quand on prend le temps de réponse comme critère de performance, l'ordinateur qui exécute la même tâche en un minimum de temps est le plus performant.

Pour estimer les performances d'un système, deux types de méthodes s'offrent [BenL96]: les méthodes basées sur la simulation et les méthodes statiques. Les méthodes basées sur la simulation consistent à exécuter le programme décrivant la tâche que ce système réalise pour des données précises. Elles sont lentes et leurs résultats sont sensibles aux données avec lesquelles ce programme a été simulé. La sensibilité à ces données peut influencer l'exploration des meilleures et des pires performances. En plus, lors des premières phases de la conception d'un système, ces méthodes ne s'appliquent pas tant qu'un modèle de celui-ci n'est pas développé. Le développement d'un tel modèle prend généralement beaucoup de temps.

Les méthodes statiques d'estimation des performances d'un système se basent sur une analyse globale de la structure du programme correspondant à la tâche à réaliser par celui-ci. Elles sont plus rapides que les méthodes mentionnées précédemment [BenL96]. En plus, nous pouvons les appliquer lors des premières phases de la conception d'un système et après la fabrication de celui-ci. C'est à ce type de méthodes que nous nous intéressons dans ce mémoire.

Pour améliorer les performances d'un système, deux possibilités s'offrent. La première possibilité est l'amélioration des performances de ses composants. Par exemple, dans le cas d'un ordinateur, nous pouvons réduire son temps de réponse en augmentant la vitesse de traitement de son processeur. Notons que l'amélioration des performances des composants d'un système n'implique pas toujours l'amélioration des performances de celui-ci. La deuxième possibilité de l'amélioration des performances d'un système est l'amélioration de son architecture.

Dans certains cas, il est souhaitable et, parfois même, nécessaire de réduire le temps de traitement requis par certaines tâches. Afin d'y arriver, nous pouvons augmenter le nombre de composants du système qui les traite. Cependant, il est possible qu'au lieu d'accélérer le traitement d'une tâche, l'augmentation du nombre de composants d'un système le ralentisse. Quand on augmente le nombre de composants d'un système, plusieurs sous-tâches de la tâche originale peuvent être traitées en parallèle par le même type de composants. Dans ce cas, nous pouvons dire que le système est un système parallèle.

Un système est dit système parallèle non seulement quand il contient plusieurs composants de même type, mais aussi nous pouvons l'appeler système parallèle si plusieurs tâches peuvent être traitées en parallèle sur un même composant de celui-ci. Pour éclaircir cette notion, prenons l'exemple d'un ordinateur. Donc, nous pouvons dire qu'un ordinateur est parallèle si son unité de traitement (processeur) est de type parallèle (e.g., processeur de type "Very Long Instruction Word") ou si on augmente le nombre de processeurs, puisque dans les deux cas, nous pouvons traiter simultanément plusieurs tâches sur le même type de composants qui est le type processeur.

Comme exemple de systèmes parallèles, nous citons les ordinateurs parallèles de type "Single Instruction Stream over Multiple Data Streams" (SIMD). Pour ce type d'ordinateurs,

plusieurs unités de traitement sont supervisées par la même unité de contrôle. Toutes ces unités reçoivent la même instruction diffusée par l'unité de contrôle, mais opèrent sur des données provenant de flux de données distincts. Comme exemple de ce type d'ordinateurs destiné pour des applications réelles, nous citons l'ordinateur *PULSE V1* "Parallel Ultra Large Scale Engine Version 1".

L'ordinateur *PULSE V1* se compose de quatre processeurs parallèles de type "Very Long Instruction Word" (VLIW) ayant des unités fonctionnelles de type "pipeline". Il est optimisé pour le traitement des applications dans le domaine du traitement d'images [MarK98]. Pour plus de détails sur cet ordinateur, le lecteur peut consulter la documentation technique de celui-ci à la référence [PULSEV1]. Un des besoins au début de la conception de cet ordinateur est l'estimation des performances de celui-ci. Ce besoin a donné la naissance des travaux de ce mémoire.

2- Objectifs et méthodologies

L'objectif de ce mémoire est de développer un outil informatique pour estimer les performances de systèmes parallèles de type SIMD et de l'appliquer à l'estimation des performances de l'ordinateur *PULSE V1*.

Pour développer cet outil, nous implémentons une méthode statique. Les algorithmes de cette méthode ont besoin d'un graphe de contrôle et de flot de données. Ce dernier est une représentation intermédiaire du programme à exécuter sur le système dont on veut estimer les performances. Dans ce mémoire, ce programme est décrit en langage de programmation C.

Pour traduire un programme à exécuter sur un système parallèle de type SIMD en un graphe de contrôle et de flot de données, nous avons besoin d'un compilateur parallèle. Dans notre cas, nous ne disposons pas de compilateur parallèle.

Comme nous l'avons mentionné à la Section 1 et nous allons voir aux prochains chapitres, la programmation d'un système parallèle de type SIMD se caractérise par l'exécution d'un seul programme sur les processeurs de ce système sous le contrôle d'une seule unité de commande

appelée contrôleur; tous les processeurs reçoivent la même instruction diffusée par le contrôleur, mais opèrent sur des données provenant de flux de données distincts. Par conséquent, en supposant que les processeurs de ce système sont identiques et que les communications inter-processeurs sont uniformes, l'estimation de ses performances revient à estimer les performances d'un de ses processeurs. Notons que ces derniers peuvent être des processeurs parallèles de type VLIW ayant des ressources de type "pipeline".

Pour traduire le programme à exécuter sur un processeur en un graphe de contrôle et de flot de données, nous utilisons un compilateur pour ANSI C appelé *LCC*. L'utilisation de ce compilateur n'est pas immédiate. Pour cela, nous apportons des modifications et des extensions à celui-ci.

Comme nous l'avons mentionné, l'outil que nous développons est une implémentation d'une méthode statique. L'utilisation de certains de ses algorithmes n'est pas directe. Donc, des adaptations de ceux-ci sont requises.

3- Organisation du mémoire

Ce mémoire est organisé de la façon suivante. Dans le premier chapitre, nous donnons une introduction au traitement parallèle et nous situons les systèmes parallèles de type SIMD. Le Chapitre 2 présente une méthode statique d'estimation des performances d'un système. La présentation du compilateur *LCC* ainsi que les modifications et les extensions que nous y apportons font l'objet du Chapitre 3. Afin de faciliter le partitionnement d'un ensemble de boucles imbriquées sur les processeurs d'un système parallèle de type SIMD tel que l'exécution des programmes résultants ne requiert par de communications interprocesseurs, nous donnons une synthèse de la littérature à ce sujet dans le Chapitre 4. Au Chapitre 5, nous présentons une méthode pour estimer les performances des machines SIMD et l'outil d'estimation de performance que nous avons développé. L'application de cet outil à l'estimation des performances d'une machine réelle de type SIMD y est examinée. Finalement, nous présentons nos conclusions et des avenues de recherche qui découlent de ce travail dans le Chapitre 6.

Chapitre 1

Introduction au traitement parallèle

Le traitement parallèle consiste à exécuter simultanément plusieurs tâches d'une application afin de réduire le temps total d'exécution. Le recours au traitement parallèle est dû aux facteurs suivants: le besoin de réduire le temps de traitement des applications dans plusieurs domaines et le fait que la vitesse des composants est limitée par un ensemble de contraintes physiques.

La naissance de l'idée du traitement parallèle dans le domaine des sciences informatiques a généré des champs de recherche qui sont actifs à l'échelle mondiale depuis les années 1940. Parmi ces champs, nous trouvons la conception des machines parallèles et des langages de programmation parallèles ainsi que le développement et l'analyse des algorithmes parallèles.

Ce chapitre introduit quelques notions de base utilisées dans le domaine du traitement parallèle. Il se compose de cinq sections. Dans la section 1, nous présentons une classification des ordinateurs. La section 2 examine la topologie des réseaux d'interconnexion qui permettent de relier entre eux les processeurs d'une machine parallèle. Nous poursuivons à la section 3 avec quelques mesures de performance des algorithmes parallèles. Dans la section 4, nous traitons de certains aspects de la programmation parallèle. Enfin, la section 5 compare les machines de type "Single Instruction Stream over Multiple Data Streams" avec les machines de type "Multiple Instruction Streams over Multiple Data Streams", introduites à la section 1.

1.1- Classification des ordinateurs

La diversité des machines parallèles ajoute une nouvelle dimension à la complexité de la conception des algorithmes parallèles. En effet, devant un problème donné, outre la question du choix d'un algorithme efficace, se pose aussi celle d'une architecture convenable.

Pour restreindre le domaine du choix, au niveau de l'architecture, une classification des ordinateurs s'impose. Dans cette section, nous présentons la classification de Flynn et ses dérivés.

1.1.1- Classification de Flynn

C'est la classification la plus populaire. Elle est basée sur le concept suivant: un ou plusieurs ("Single" ou "Multiple") flot(s) d'instructions agissent sur un ou plusieurs flot(s) de données [Flyn72]. Ceci mène à quatre classes d'ordinateurs (Figure 1.1):

"Single Instruction Stream over a Single Data Stream" (SISD): Une seule instruction est exécutée à chaque pas et n'agit que sur un seul ensemble de données. Les machines séquentielles, basées sur le modèle de Von Neumann, sont dans cette classe.

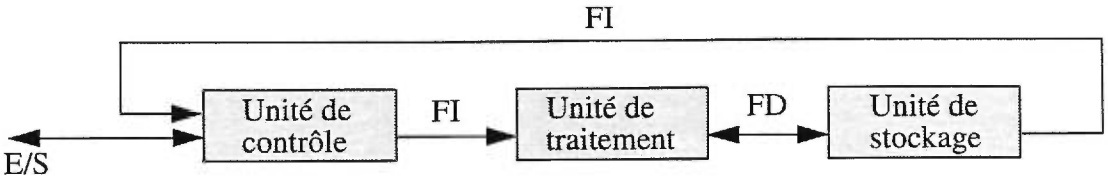
"Single Instruction Stream over a Multiple Data Streams" (SIMD): Un seul type d'instructions est exécuté à la fois sur des données différentes. Un exemple de machines commerciales pour cette classe est la machine Illiac IV.

"Multiple Instruction Streams and a Single Data Stream" (MISD): Plusieurs instructions peuvent être exécutées simultanément sur un seul ensemble de données. Cette classe est nommé réseaux systoliques. Un exemple de machines commerciales est Iwarp.

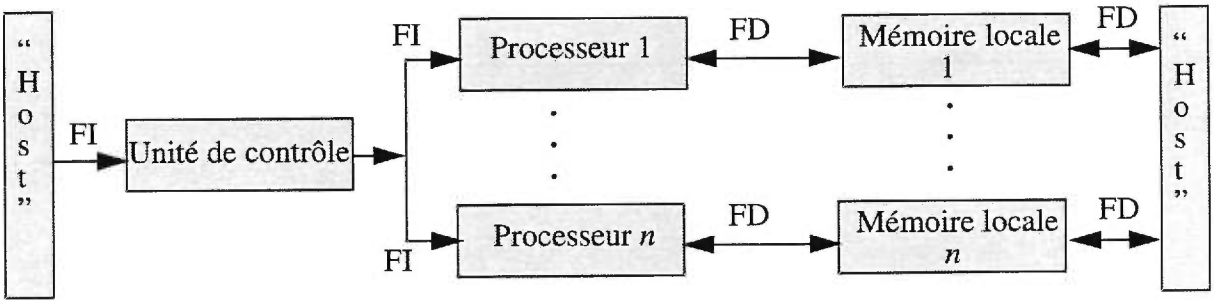
"Multiple Instruction Streams over Multiple Data Streams" (MIMD): Plusieurs instructions peuvent être exécutées sur des données différentes. La machine DASH est un exemple de machines commerciales pour cette famille.

1.1.2- Autres classifications

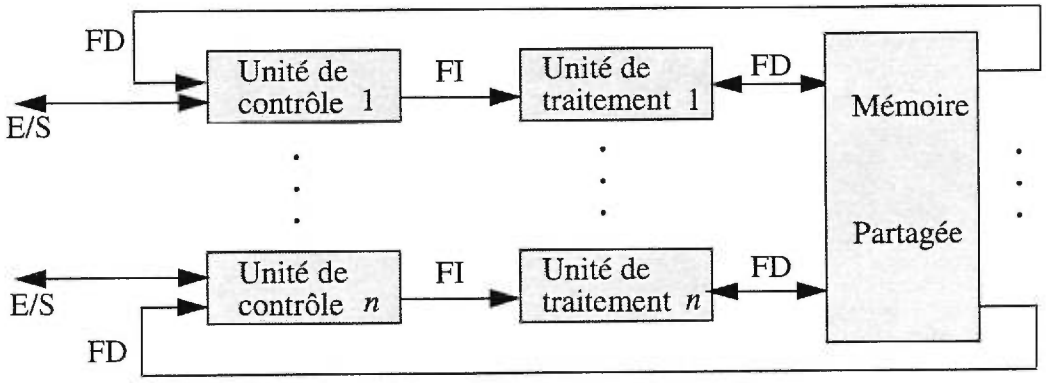
D'autres classes peuvent être dérivées de celles de Flynn. Par exemple, les multiprocesseurs à mémoire partagée, les multiprocesseurs à échange de messages, et les Multi-SIMD sont des sous classes de la famille MIMD [Hwan84] et [Hwan93].



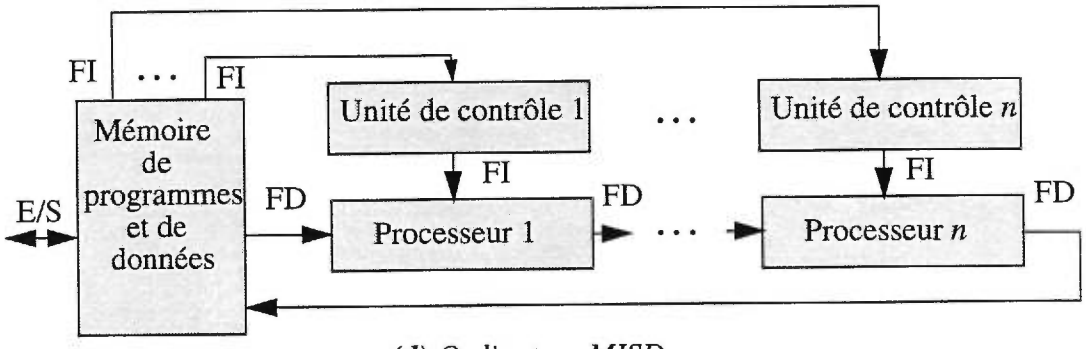
(a) Ordinateur SISD.



(b) Ordinateur SIMD à mémoire distribuée.



(c) Ordinateur MIMD à mémoire partagée.



(d) Ordinateur MISD.

FI: Flot d'instructions
FD: Flot de données
E/S: Entrées/Sorties

Figure 1.1 Classification de Flynn.

Dans le cas des Multi-SIMD, plusieurs instructions, qui peuvent être de types différents, s'exécutent sur plusieurs tableaux de processeurs. Chaque tableau de processeurs appartient à la classe SIMD.

1.2- Topologies des réseaux d'interconnexion

Les réseaux d'interconnexion sont utilisés dans les systèmes multiprocesseurs pour relier les processeurs entre eux ou les processeurs à la mémoire. Dans cette section, nous présentons certaines caractéristiques et paramètres des réseaux d'interconnexion.

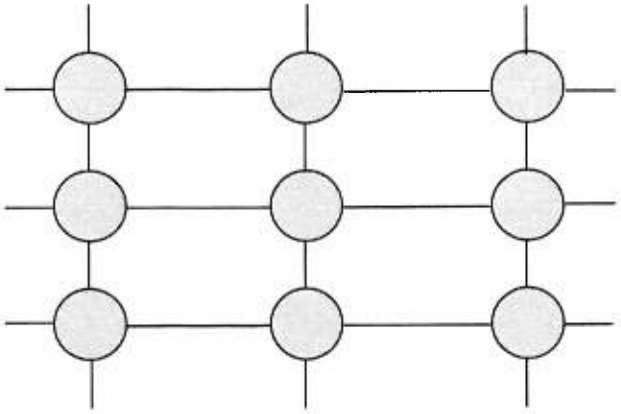
1.2.1- Certaines caractéristiques des réseaux d'interconnexion

La topologie: Elle définit la structure suivant laquelle les noeuds (processeurs) sont interconnectés. Des exemples de topologies (Figure 1.2) sont les mailles, les réseaux linéaires et les hypercubes. Ces types de topologies sont parmi les plus populaires. Par exemple, dans le cas d'un réseau linéaire, les noeuds peuvent être des réseaux dont la topologie est une maille, un hypercube, etc.

Le mode de fonctionnement: Il est synchrone ou asynchrone.

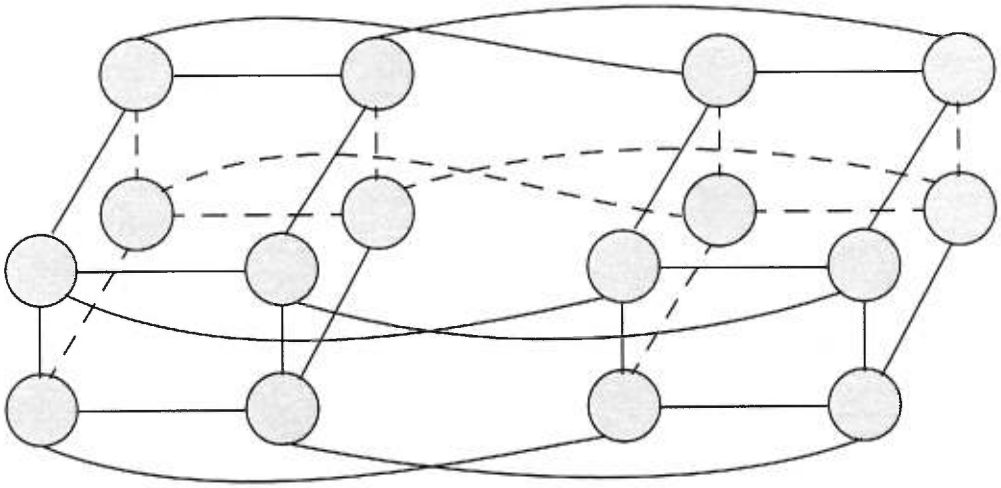
Les liens physiques de communication: Ils peuvent être statiques ou dynamiques. Dans le cas statique, les liens sont fixés une fois pour toutes lors de la conception du réseau; ce type est souhaitable si nous connaissons à l'avance la nature des communications. Par contre, si nous n'avons pas une idée de la manière de communiquer, ce qui est le cas pour des applications générales, le type dynamique sera la solution. Ce dernier type a un désavantage qui est le temps de configuration du réseau.

Le mode de contrôle: Il est centralisé (cas de la classe SIMD), réparti (cas de la famille MIMD) ou hybride.



(a) Maille à deux dimensions ayant neuf processeurs.

(b) Réseau linéaire.



(c) Hypercube.

Figure 1.2 Exemples de topologies.

1.2.2- Paramètres importants

Certains paramètres du graphe représentant la topologie, sont particulièrement importants:

Le diamètre: C'est la distance maximale entre les paires de noeuds, où la distance est le nombre minimum de liens à traverser pour aller d'un noeud de la paire vers l'autre. Par exemple, pour un réseau linéaire de p processeurs, le diamètre vaut (p-1).

La connectivité: Elle peut être définie au sens des arêtes ou des noeuds. La connectivité au sens des arêtes (des noeuds) est le nombre minimum d'arêtes (de noeuds) à enlever pour que le réseau devienne non-connexe. Une grande valeur de la connectivité est désirée, car elle rend les communications rapides et augmente la fiabilité. La connectivité pour un réseau linéaire de p processeurs vaut 2 si les liens sont bidirectionnels, et 1 si ces derniers sont unidirectionnels.

Le degré: C'est le plus grand élément de l'ensemble des degrés des noeuds. Le degré d'un noeud est le nombre d'arêtes incidentes sur celui-ci. Au point de vue matériel, le degré d'un noeud représente le nombre de ports d'entrées et de sorties de celui-ci. En conséquence, le degré d'un noeud est en relation directe avec son coût. D'où, pour réduire le coût d'un noeud, le degré de celui-ci doit être le plus petit possible tout en satisfaisant le besoin de performances. Pour un réseau linéaire de processeurs, le degré vaut 2.

1.3- Mesures de performances des algorithmes parallèles

Dans l'évaluation de la qualité d'un algorithme parallèle, les critères suivants sont souvent utilisés: le temps d'exécution, l'accélération et l'efficacité. Nous définissons ces trois critères aux sections suivantes.

1.3.1- Le temps d'exécution

C'est une mesure importante car la réduction du temps de calcul est la raison principale du traitement parallèle. Le temps d'exécution est défini comme le temps entre l'instant où le premier processeur commence à travailler et celui où tous les processeurs finissent leur travail.

L'efficacité d'un algorithme parallèle est basée sur la comparaison du temps d'exécution de l'algorithme parallèle et celui du meilleur algorithme séquentiel. Cette comparaison utilise la notion d'accélération.

1.3.2- Accélération

Pour un problème donné, soit T_s le temps de son exécution séquentielle et T_p celui de son exécution parallèle en utilisant p processeurs.

L'accélération S_p est le rapport entre le temps d'exécution du meilleur algorithme séquentiel et celui de l'algorithme parallèle. Soit donc $S_p = \frac{T_s}{T_p}$.

La valeur de S_p est dans l'intervalle [1, p]. Durant le développement d'un algorithme parallèle, nous devons maximiser la valeur de S_p. Dans un cas parfait, S_p vaut p.

Un algorithme parallèle est divisible en deux parties: une partie parallèle et une partie séquentielle. Quand le nombre de processeurs p est grand, la partie parallèle s'exécute rapidement tandis que la partie séquentielle reste invariante. Ce phénomène a été décrit par Amdahl en 1967, et on l'a décrit plus tard par *loi d'Amdahl*. Cette loi s'énonce comme suit: $S_p \leq \frac{p}{1+(p-1)\cdot\alpha} \leq \frac{1}{\alpha}$, où α est la fraction du temps d'exécution de la partie séquentielle par rapport au temps total d'exécution de l'algorithme parallèle.

1.3.3- L'efficacité

Pour un système parallèle de p processeurs, l'efficacité E_p est définie par le rapport entre l'accélération et le nombre de processeurs comme suit: $E_p = \frac{S_p}{p}$.

L'efficacité mesure la fraction du temps où les processeurs travaillent réellement. Idéalement, E_p vaut 1. En général, la valeur de E_p est dans l'intervalle [0, 1].

1.4- Programmation parallèle

Dans cette section, nous présentons trois approches pour le choix d'un langage en vue d'implanter un algorithme sur une machine parallèle. En outre, nous spécifions quatre étapes jugées importantes pour la conception d'un algorithme parallèle.

1.4.1- Langages de programmation

L'implantation d'un algorithme sur une machine parallèle peut se faire en utilisant un langage de programmation séquentiel, l'extension d'un langage séquentiel ou encore un langage parallèle.

1.4.1.1- Langage séquentiel

L'avantage d'utilisation d'un langage séquentiel est la possibilité de continuer à utiliser des programmes séquentiel existants. Dans ce cas, l'implantation d'un algorithme sur une machine parallèle nécessite un compilateur parallélisant. Le parallélisme est alors implicite.

Un désavantage de cette approche est que le compilateur peut prendre des décisions plus naïves que celles prises par le programmeur, car seules les formes du parallélisme identifiées par ce compilateur seront exploitées.

1.4.1.2- Extension d'un langage séquentiel

Cette approche est basée sur une extension d'un langage séquentiel par une bibliothèque de fonctions adaptées au traitement parallèle sur une machine donnée. Un de ses avantages est la possibilité de développer des programmes parallèles à partir des versions séquentielles. Ceci réduit le coût et les erreurs associées au développement des programmes.

1.4.1.3- Langage parallèle

Dans ce cas, un nouveau langage de programmation est conçu. La conception d'un nouveau

langage parallèle permet au programmeur de spécifier explicitement le parallélisme. Le parallélisme est donc explicite.

Ce parallélisme explicite peut conduire à des implantations plus efficaces que celles obtenues à l'aide d'un compilateur parallélisant. En effet, le programmeur peut prendre de meilleures décisions qu'un compilateur. Néanmoins, la tâche du programmeur est, en général, difficile.

1.4.2- Conception d'algorithmes parallèles

Quatre tâches sont importantes dans la phase de conception d'un algorithme parallèle: le partitionnement, la communication, le regroupement et l'allocation. En général, chacune de ces quatre tâches est un problème NP-complet ([CofD73], [LenR78], [Chré89], [Sark89] et [ChrC95]).

1.4.2.1- Le partitionnement

C'est le partage d'un traitement en plusieurs tâches pouvant être traitées en parallèle sur un ensemble de processeurs. Ce partitionnement doit être conçu de façon à assurer un équilibre de la charge entre les processeurs. Le but de cet équilibrage de charge est la réduction du temps global de traitement.

1.4.2.2- La communication

Le partionnement peut conduire à des tâches qui doivent communiquer des informations entre elles. Ces échanges d'informations se font à travers le réseau d'interconnexion et peut avoir des impacts sur les performances.

Afin de réduire les impacts des communications des tâches sur les performances d'un algorithme parallèle, ces dernières doivent être minimisées tout en maximisant le nombre de processeurs actifs à chaque instant.

1.4.2.3- Le regroupement

Il consiste à assembler plusieurs sous-tâches en une afin d'améliorer les performances ou de réduire le coût du développement.

1.4.2.4- L'allocation

Elle consiste à déterminer le temps d'exécution des tâches, obtenues par le partionnement, en tenant compte des contraintes de précédence et de l'architecture de la machine. Notons que pendant l'allocation, la maximisation des performances doit être considérée.

1.5- Comparaison des machines SIMD et MIMD

Les machines SIMD et MIMD sont constituées d'un ensemble de processeurs interconnectés. Ces processeurs peuvent être de divers types tels que: scalaire, pipeline, superscalaire, superpipeline et "Very Long Instruction Word" (VLIW) [Hwan93]. Chacun de ces deux types de machines peut offrir des meilleures performances que l'autre pour une famille d'applications donnée. Par exemple, pour une classe d'applications dont le traitement est régulier et où il y a un parallélisme de données, les machines SIMD ont de meilleures performances. En effet, ces machines sont spécialement conçues pour ce type d'applications [FisH95]. Pour des applications dont le traitement est irrégulier, les machines MIMD ont de meilleures performances.

Dans le reste de cette section, nous donnons une brève comparaison de ces deux types de machines. Cette comparaison est basée sur les références [BerS91], [Zhen91], [BalB94], [FonD94], [Sand94], [Sieg95], [AllS96], [GanG96] et [WuSh96].

1.5.1- Les avantages des machines SIMD

Certains avantages des machines SIMD par rapport aux machines MIMD sont:

- L'écriture d'un seul programme facilite la détection des erreurs.

- Une seule copie des instructions est nécessaire. Ceci permet, probablement, la réduction de la taille de la mémoire et de son coût, et la réduction de la communication entre la mémoire primaire et la mémoire secondaire.

- La présence d'un seul compteur de programme et le besoin d'un seul décodeur réduisent le coût du matériel.

- La synchronisation des processeurs est implicite. Elle est explicite dans le cas des machines MIMD. Ce type de synchronisation nécessite un protocole de gestion des communications interprocesseurs.

- L'écriture d'un seul programme peut amener à une meilleure utilisation de la machine. L'utilisation de p processeurs d'une machine MIMD demande l'écriture de p programmes. L'écriture de p programmes peut avoir des impacts sur l'efficacité de l'utilisation de la machine MIMD, notamment à cause de la communication entre ces programmes.

1.5.2- Les avantages des machines MIMD

Certains avantages des machines MIMD par rapport aux machines SIMD sont:

- La flexibilité et l'utilisation pour des applications généralisées. L'utilisation des machines SIMD est recommandée pour des applications dont le traitement est régulier et où il y a un parallélisme de données.

- Les performances lors de l'exécution des instructions conditionnelles ("if then else") sur une machine MIMD sont meilleures que celles sur une machine SIMD. Ceci s'explique par le fait que dans le cas d'une machine MIMD, chaque processeur peut suivre, selon sa décision, une des deux branches du "if" indépendamment des décisions des autres processeurs. Dans le cas d'une machine SIMD, les processeurs ne peuvent que suivre un même type de branches (i.e., la branche "then" ou la branche "else").

- Pour les machines SIMD, les processeurs se synchronisent après le traitement de chaque instruction. Dans certains cas, cette synchronisation n'est pas nécessaire pour les machines MIMD. Par conséquent, le temps d'exécution pour une machine MIMD est moindre que celui pour une machine SIMD.

- Pour une machine SIMD, à chaque instant, les processeurs ne peuvent exécuter que des instructions de même type. Par conséquent, pour certaines applications, la conception de programmes efficaces pour une machine SIMD est plus difficile.

Chapitre 2

Méthode statique d'estimation de performance d'un système

L'objectif de ce chapitre est de donner une méthode rapide pour estimer les performances d'un système réalisant une tâche donnée. Plus précisément, cette méthode nous permet d'estimer, rapidement, le temps d'exécution du programme correspondant à cette tâche sur ce système.

Pour estimer les performances d'un système, deux types de méthodes s'offrent: les méthodes basées sur la simulation et les méthodes statiques. Les méthodes basées sur la simulation consistent à exécuter le programme décrivant la tâche que ce système réalise pour des données précises.

Les méthodes statiques d'estimation des performances d'un système se basent sur une analyse globale de la structure du programme correspondant à la tâche à réaliser par celui-ci. Comme nous l'avons expliqué dans l'introduction de ce mémoire, ces méthodes sont plus rapides que celles mentionnées précédemment. En plus, nous pouvons les appliquer lors des premières phases de la conception d'un système et après la fabrication de celui-ci.

Ce chapitre présente une méthode statique d'estimation de performance d'un système. La Figure 2.1 en décrit les étapes. À partir de la description de l'architecture de ce système sous forme d'unités fonctionnelles, de mémoires, de bus ainsi que des interconnexions, et du programme décrivant une tâche à réaliser, traduit par la suite en un graphe de contrôle et de flot de données (GCFD), nous dérivons des bornes inférieures et supérieures sur les performances. Nous désignons au long de ce chapitre par performance le temps d'exécution.

Cette méthode est basée sur des techniques d'ordonnancement et d'allocation. Les problèmes d'ordonnancement et d'allocation sont, en général, NP-complet ([LenR78], [McFP90]

et [ChrC95]). Afin de résoudre ceux-ci, nous utilisons des algorithmes voraces.

Le reste de ce chapitre est organisé comme suit. Après avoir examiné le modèle d'architectures utilisé dans la méthode mentionnée précédemment, ainsi que le modèle GCFD dans les deux premières sections, nous présentons l'ensemble des algorithmes pour la détermination des bornes inférieure et supérieure sur les performances d'un système.

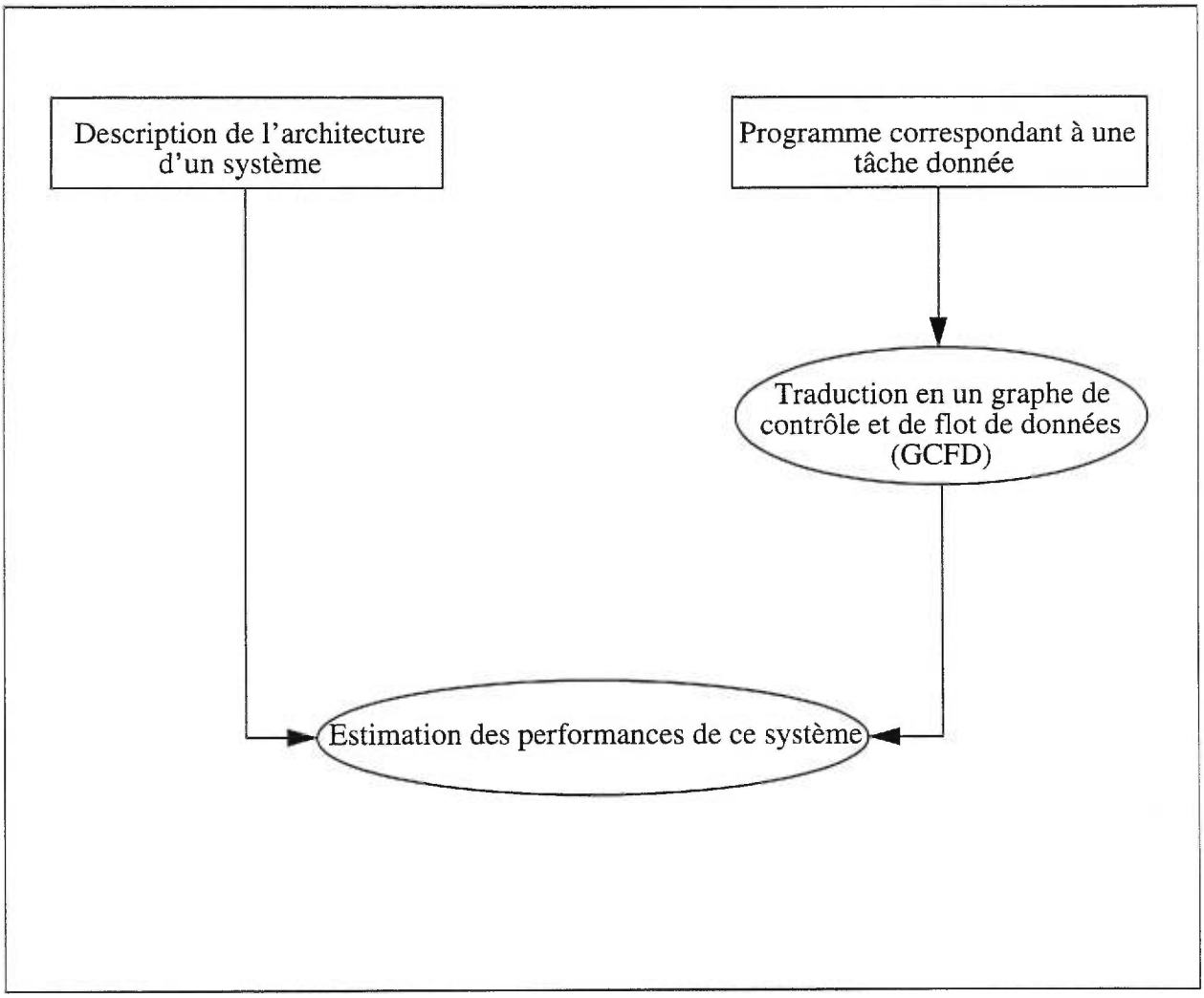


Figure 2.1 Description d'une méthode statique d'estimation des performances d'un système.

2.1- Modèle d'architectures

La Figure 2.2 présente le modèle d'architecture utilisée dans ce travail. Ce modèle est constitué de trois éléments essentiels: une mémoire, des unités fonctionnelles ainsi que des bus. Il y a p ports pour la mémoire, k types différents d'unités fonctionnelles, b_s bus source et b_d bus destination. Les unités fonctionnelles de même type peuvent être de type "pipeline" et en plusieurs exemplaires.

Le modèle d'architecture présenté à la Figure 2.2 est basé sur une architecture à bus. Les raisons de notre choix de ce modèle sont:

- Ce modèle d'architecture permet de modéliser l'architecture des processeurs d'un ordinateur parallèle en cours de conception appelé *PULSE V1* destiné pour des applications réelles. Pour plus de détails sur cet ordinateur, le lecteur peut consulter la documentation technique de celui-ci à la référence [PULSEV1]. Notons que le besoin d'estimer les performances de cet ordinateur est la motivation principale de ce travail.

- Les architectures à bus permettent la spécification d'une large famille d'architectures allant des machines séquentielles aux machines massivement parallèles.

- Les architectures à bus sont très utilisées dans les architectures modernes.

- Les architectures à bus permettent la réduction des surfaces des circuits en réduisant le nombre de multiplexeurs requis.

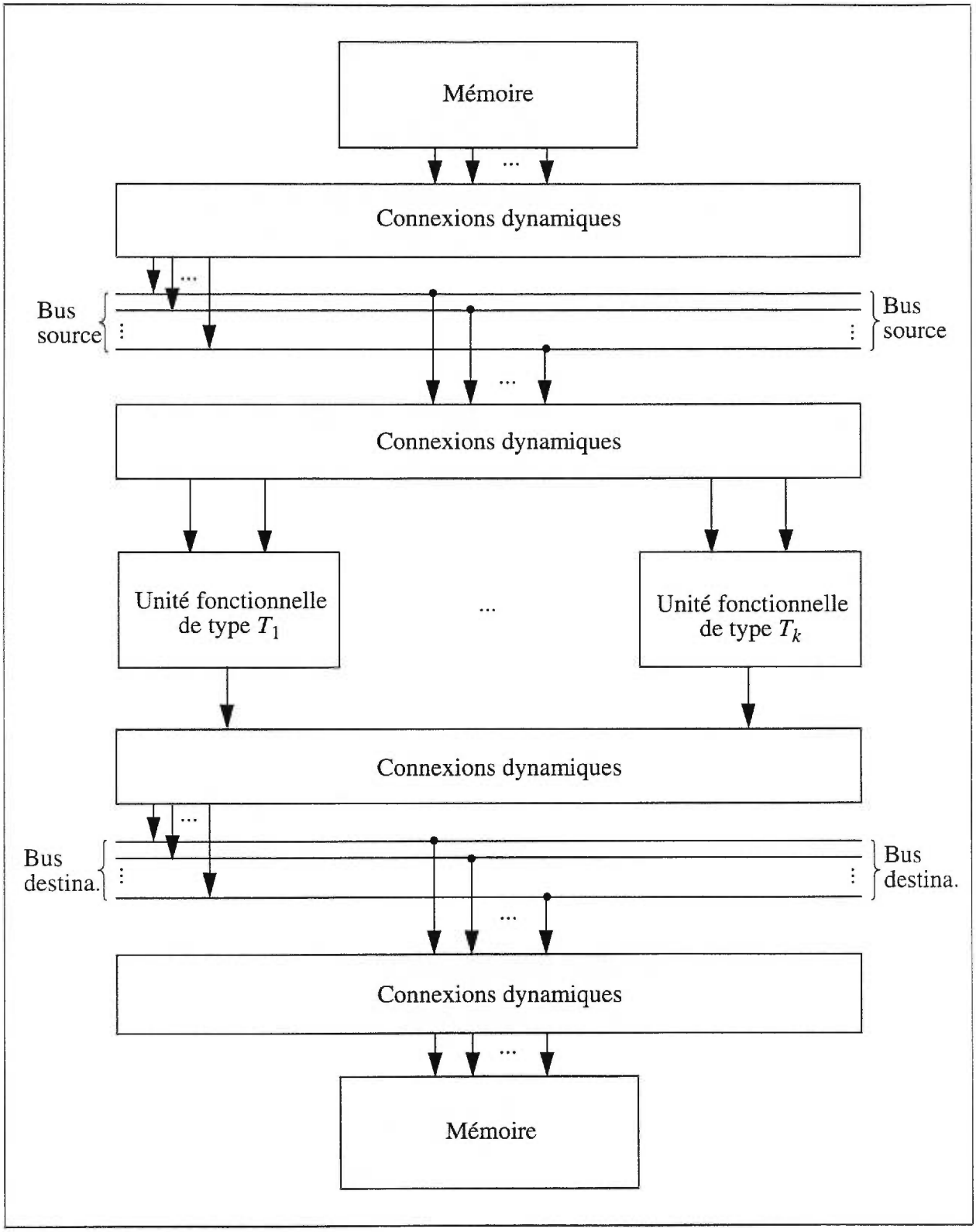


Figure 2.2 Modèle d'architecture utilisée.

2.2- Le modèle graphe de contrôle et de flot de données: GCFD

Un graphe de contrôle et de flot de données (GCFD) est une représentation intermédiaire d'un programme source ([GajD92] et [GajR94]. Il se compose de deux types de graphes: un graphe de flot de contrôle (GFC) et un graphe de flot de données (GFD), que nous allons présenter dans les paragraphes suivants. Dans ce travail, ce programme est écrit dans le langage C avec les restrictions suivantes: pas d'appel de fonctions et pas d'instructions *switch*. Notre choix de ce langage ainsi que ces restrictions découlent des besoins du projet *PULSE* (pour plus de détails sur ce projet, voir la référence [PULSEV1]) qui consiste à concevoir un ordinateur parallèle appelé *PULSE V1* destiné pour des applications réelles comme nous l'avons déjà mentionné dans la section précédente. Notons que le choix du langage pour écrire un programme source dépend de plusieurs facteurs à savoir par exemple: l'existence de code déjà développé dans le langage à choisir, la familiarisation du programmeur avec celui-ci, le fait que ce langage est un langage de haut-niveau et qu'il se caractérise par une popularité d'utilisation (comme par exemple le langage C), et l'existence de compilateurs efficaces pour traduire le programme écrit dans ce langage en un GCFD.

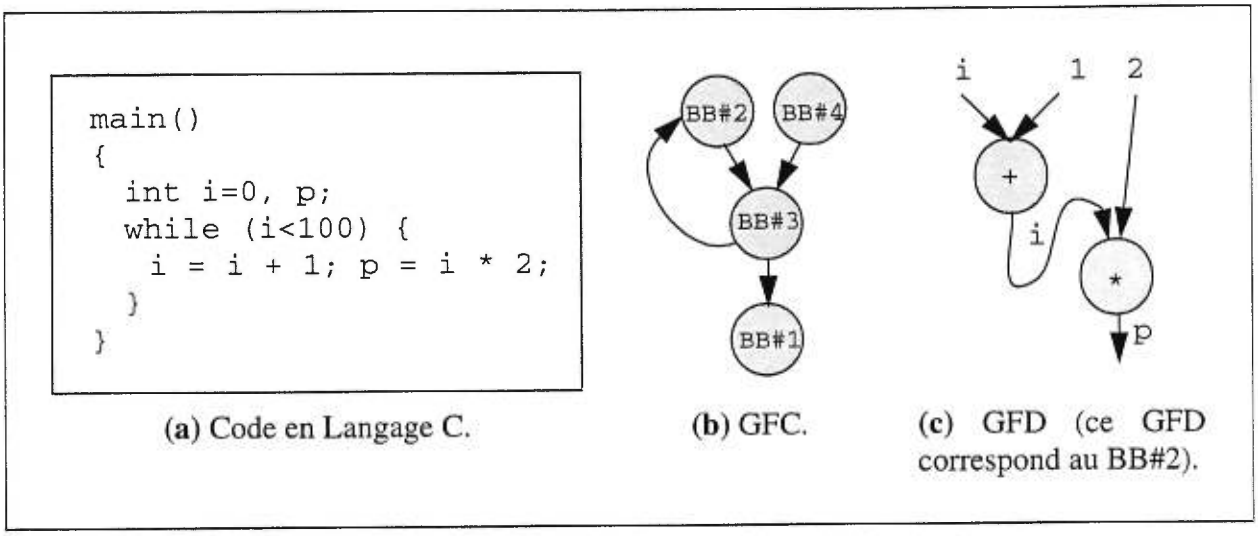
Un GFC (voir un exemple à la Figure 2.3 (b)) est un graphe orienté dont les noeuds sont des GFDs; ces derniers sont définis dans le paragraphe suivant. Ses arcs représentent les dépendances entre les GFDs.

Un GFD (voir un exemple à la Figure 2.3 (c)) est un graphe orienté dont les noeuds sont des opérations atomiques (e.g., addition). Les arcs de ce GFD constituent les dépendances de données entre les opérations atomiques. Le poids, p, de chaque arc est le temps d'exécution de son noeud source; en conséquence, l'exécution du noeud destination de cet arc ne peut commencer qu'à un instant t_d tel que $t_d \geq (t_s + p)$, où t_s est le temps de début de l'exécution de ce noeud source.

Dans la littérature, un GFD est appelé aussi un bloc de base (BB). Notons qu'un BB peut ne pas être un DFG, car il peut représenter des dépendances de contrôle. Pour la simplicité, nous utilisons, au long de ce chapitre, l'appellation bloc de base (BB) pour désigner un graphe de flot de données au lieu de l'appellation GFD.

Quand le programme source contient un traitement itératif (i.e., présence de boucles), la

méthode que nous présentons dans ce chapitre requiert la spécification des nombres minimal et maximal d'itérations de chacune de ces boucles. Afin de spécifier ces nombres, des annotations du programme source doivent être faites. Comme nous allons voir dans la section suivante, les nombres minimal et maximal d'itérations de chaque boucle aident à la détermination, respectivement, d'une borne inférieure et d'une borne supérieure sur le temps d'exécution du programme source.



(a) Code en Langage C. (b) GFC. (c) GFD (ce GFD correspond au BB#2).

Figure 2.3 Exemple d'un graphe de contrôle et de flot de données.

2.3- Bornes inférieure et supérieure sur le temps d'exécution d'un programme

Pour toutes les valeurs possibles des données d'un programme source P, si le temps de son exécution est dans l'intervalle $[Binf, Bsup]$ (où $Binf \leq Bsup$), alors $Binf$ et $Bsup$ sont appelées, respectivement, borne inférieure et borne supérieure sur le temps d'exécution de P.

Une fois que le programme source est traduit en un graphe de contrôle et de flot de données (GCFD), l'estimation des bornes inférieure et supérieure sur le temps d'exécution revient à déterminer la longueur du plus court et du plus long chemins dans ce GCFD. Pour calculer la longueur du plus court et du plus long chemins dans le graphe GCFD, nous devons tout d'abord déterminer, respectivement, une borne inférieure et une borne supérieure sur le temps d'exécution de chacun de ses blocs de base (BBs).

Pour la méthode statique d'estimation de performance d'un système, présentée dans ce

chapitre, nous supposons que l'exécution des blocs de base est séquentielle, et qu'on n'applique pas de techniques d'optimisation sur chacun de ces blocs une fois le GCFD est généré; pour plus de détails sur ces techniques, le lecteur peut consulter la référence [DeMi94]. En plus, quand le programme source correspond à un traitement itératif, nous supposons qu'on n'applique pas de techniques d'accélération de boucles (par exemple la réécriture de boucles) durant la génération du GCFD. En conséquence, le nombre d'itérations minimal et le nombre d'itérations maximal de celles-ci avant et après la génération du GCFD sont les mêmes.

En terme pratique, des valeurs serrées pour une borne inférieure (*Binf*) et une borne supérieure (*Bsup*) sur le temps d'exécution d'un programme permettent non seulement d'avoir une bonne estimation des performances d'un système, mais elles permettent aussi de vérifier si des performances requises peuvent être obtenues avec les ressources disponibles de celui-ci. En effet, supposons par exemple qu'on requiert que le temps maximal d'exécution d'un programme sur un système soit *Tmax*. Si $Tmax \geq Bsup$, alors on conclut que la performance requise pourrait être atteinte avec les ressources de ce système; par contre, si $Tmax < Binf$, nous pourrons conclure que cette performance ne peut être satisfaite avec les ressources de celui-ci.

Le reste de cette section présente d'abord les algorithmes "As Soon As Possible" (ASAP) et "As Late As Possible" (ALAP); ces derniers sont utilisés dans les algorithmes que nous allons présenter dans les sous-sections qui suivent. La deuxième sous-section présente un algorithme pour le calcul d'une borne inférieure sur le temps d'exécution d'un bloc de base (BB). À la troisième sous-section, nous présentons un algorithme pour calculer une borne supérieure sur le temps d'exécution d'un BB. La quatrième sous-section présente un algorithme pour la détermination d'une borne inférieure sur le temps d'exécution du graphe de contrôle et de flot de données (GCFD). Finalement, dans la cinquième sous-section, nous présentons un algorithme pour déterminer une borne supérieure sur le temps d'exécution du GCFD. Tous ces algorithmes se caractérisent par une faible complexité en temps de calcul (i.e., complexité polynomiale).

2.3.1- Les algorithmes ASAP et ALAP

L'algorithme ASAP consiste à ordonnancer les noeuds d'un graphe le plus tôt possible tel que la contrainte des dépendances inter-noeuds est respectée. Cette contrainte est définie comme suit. Si N_1 et N_2 sont deux noeuds d'un graphe tels que N_2 dépend de N_1 et si T_1 et T_2 sont les temps

d'ordonnancement de N_1 et de N_2, respectivement, alors $T_2 \geq T_1 + d$, où d est le délai du noeud N_1. Cette définition est considérée tout au long de ce chapitre. La contrainte des ressources dans cet algorithme est ignorée. L'algorithme au complet est donnée à la Figure 2.4. Pour plus d'informations sur cet algorithme, le lecteur peut se référer à [DeMi94] et à [WalC95].

L'algorithme ALAP peut être dérivé de l'algorithme ASAP. Ceci en commençant l'ordonnancement à partir du dernier noeud du graphe à l'instant λ, où λ est la latence produite par l'algorithme ASAP. Pour plus de détails sur l'algorithme ALAP, le lecteur peut consulter [DeMi94] et [WalC95].

Entrée: Un graphe $G(V, E)$. V et E sont respectivement l'ensemble des noeuds et des arcs de G.
Sortie: La latence λ de l'ordonnancement des noeuds du graphe G. Elle est définie comme suit: $\lambda = max_{(i,j)}(|t_i - t_j| + 1)$, où t_i et t_j sont, respectivement, les instants d'ordonnancement des noeuds i et j de ce graphe.
Début /* v_{in} et v_{out} sont deux noeuds de syncronisation. */

- Ordonnancer v_{in} au temps $T_{vin} = 1$.
- Tant que (v_{out} n'est pas ordonnancé) faire
  - Séléctionner un noeud v_i de V tel que ses prédécesseurs sont tous ordonnancés.
  - Ordonnancer v_i au temps T_{v_i} tel que $T_{v_i} = max_{j:(v_j, v_i) \in E}(T_{v_j} + d_j)$, où d_j est le délai de v_j.
- $\lambda = max_{(i,j)}(|t_i - t_j| + 1) = T_{vout} - T_{vin} + 1 = T_{vout}$.

Fin.

Figure 2.4 L'algorithme "As Soon As Possible".

2.3.2- Borne inférieure sur le temps d'exécution d'un bloc de base

Le poids de chaque noeud (opération atomique) du graphe correspondant à un bloc de base est égal au temps nécessaire à l'exécution de cette opération. Une borne inférieure triviale sur le temps d'exécution d'un bloc de base est la longueur de chemin critique (i.e., la longueur de plus long chemin) de celui-ci. Pour calculer la longueur de plus long chemin dans le graphe correspondant à un bloc de base, on considère que le poids de chaque arc est égal au poids du noeud

source de celui-ci. Dans le cas où il y a une contrainte de ressources, plusieurs algorithmes pour le calcul d'une borne inférieure sur le temps d'exécution d'un bloc de base ont été proposés [Fren82], [JainP92], [OhmJ92], [RimJ94], [LinY97]. Par exemple, l'algorithme de Rim et Jain, [RimJ94], est un algorithme vorace. L'avantage de celui-ci par rapport aux autres algorithmes est qu'il produit une solution de compromis entre la rapidité d'exécution et l'optimalité de la solution. L'idée de base dans cet algorithme est la suivante. On ignore la contrainte des dépendances mais on considère celle des ressources et le fait que chaque opération o_i doit être ordonnancée au moment le plus tôt possible T_{o_i} qui soit supérieur ou égal au temps d'ordonnancement de o_i selon l'algorithme ASAP. L'absence de la contrainte des dépendances a pour conséquence qu'un noeud peut être ordonnancé avant ses prédécesseurs. La contrainte des bus n'est pas considérée. Une fois que toutes les opérations sont ordonnancées, l'algorithme calcule la valeur maximale V_{max} de $(T_{o_i} - ALAP_{o_i})$ en variant o_i, où $ALAP_{o_i}$ est le temps d'ordonnancement de o_i selon l'algorithme ALAP. Finalement, la borne inférieure recherchée pour le temps d'exécution d'un bloc de base est la somme de la valeur V_{max} et de la longueur de chemin critique de celui-ci.

Afin de traiter des architectures similaires au modèle d'architecture que nous avons présenté à la Section 2.1, nous avons adapté l'algorithme de Rim et Jain, [RimJ94], comme il est montré à la Figure 2.5. L'algorithme ainsi obtenu (voir Figure 2.5) est implémenté dans l'outil *C_PerfEstim*. Nous présenterons cet outil au Chapitre 5.

Pour faciliter la compréhension de l'algorithme présenté à la Figure 2.5, nous donnons les définitions et les notations suivantes:

- $ASAP_{o_i}$ et $ALAP_{o_i}$ sont les dates d'ordonnancement de l'opération o_i. Ces dates sont données, respectivement, par les algorithmes *ALAP* et *ASAP*.
- Pour les unités fonctionnelles
  - k est le nombre de types différents d'unités fonctionnelles.
  - S_j est l'ensemble des opérations o_i qui requièrent une unité fonctionnelle de type T_j où $j = 1, \ldots, k$.
  - Q_{T_j} est la liste des unités fonctionnelles de type T_j où $j = 1, \ldots, k$.
- Pour la mémoire

- S_{P_r} est la liste des opérations qui requièrent un port en lecture.
- S_{P_w} est la liste des opérations qui requièrent un port en écriture.
- P_r est le nombre de ports en lecture. Certains de ces ports peuvent être en écriture.
- P_w est le nombre de ports en écriture. Certains de ces ports peuvent être en lecture.
- *Borne* = (P_r + P_w) - le nombre de ports en lecture et en écriture. *Borne* est le nombre total maximal des lectures de la mémoire et des écritures dans celle-ci qu'on peut faire pendant un cycle donné.
- *RplusW* est le nombre total des lectures de la mémoire et des écritures dans celle-ci pendant un cycle donné. Ce nombre ne doit pas dépasser *Borne*.

- *CC* est la longueur de chemin critique d'un bloc de base. C'est un nombre entier. Il est en cycles d'horloge.
- $Boite_x$ est l'ensemble des opérations qui ont un temps d'ordonnancement selon l'algorithme *ASAP* égal à x où $x = 1, \ldots, CC$.
- $Boite_{temp}$ est une liste temporaire. Elle contient les opérations à ordonnancer pour un cycle donné. Elle est en ordre croissant des valeurs de $ALAP_{o_i}$. On consomme les éléments de cette liste selon l'ordre croissant des valeurs de $ALAP_{o_i}$.

Entrées: Un bloc de base (*BB*) et la description de l'architecture d'un système.
Sortie: Une borne inférieure (*Binf*) sur le temps d'exécution du bloc de base en entrée.
Début
/* Initialisation. */

- Insérer les unités fonctionnelles de type T_j dans la liste Q_{T_j}. $j = 1, ..., k$.
- Pour la mémoire, initialiser le nombre de ports en lecture P_r et le nombre de ports en écriture P_w.
- Faire: $Borne = (P_r + P_w)$ - le nombre de ports en lecture et en écriture.
- Trouver le $ASAP_{o_i}$ et le $ALAP_{o_i}$ pour chaque opération o_i du *BB*.
- Distribuer les opérations o_i du *BB* sur les listes $S_{j=1, ..., k}$, S_{Pr} et S_{Pw}. Ces listes sont en ordre croissant des valeurs des $ALAP_{o_i}$.
- Calculer la longueur de chemin critique *CC* du *BB*.
- Créer *CC* boites numérotées de 1 à *CC*. Mettre dans chaque boite, $Boite_{x=1, ...CC}$, les opérations dont les *ASAP* valent *x* (on commence l'ordonnancement au cycle un au lieu du cycle zéro).
- Initialiser le compteur du temps *C* à 0.
- Initialiser *x* à 0 et la liste $Boite_{temp}$ à vide.

/* Ordonnancement des opérations du BB. */

- Tant que (tous les noeuds ne sont pas ordonnancés) faire
  - $C = C + 1$ et $RplusW = 0$.
  - Si $x < CC$, faire $x = x + 1$ et ajouter le contenu de $Boite_x$ à celui de $Boite_{temp}$.

  /* Ordonnancement des opérations *op* de $Boite_{temp}$ sur une unité fonctionnelle de type T_j.*/
  Pour j=1 à *k* faire
  Tant que ($Boite_{temp} \neq \varnothing$ et $op \in (Boite_{temp} \cap Q_{T_j})$ et $Q_{T_j} \neq \varnothing$) faire
  - Supprimer une unité fonctionnelle R_{T_j} de Q_{T_j}.
  - Ordonnancer *op* sur R_{T_j} au cycle *C* et retourner la date de disponibilité de celle-ci sous forme $<R_{T_j}$, disponible_quand>.
  - Mettre $<R_{T_j}$, disponible_quand> dans la liste des dates de disponibilité des ressources.
  - Supprimer *op* de $Boite_{temp}$ et de toutes les listes S_j.

  /* Ordonnancement des opérations *op* de $Boite_{temp}$ sur un port mémoire en lecture.*/
  Tant que ($Boite_{temp} \neq \varnothing$ et $op \in (Boite_{temp} \cap S_{P_r})$ et $P_r \neq 0$ et $RplusW < Borne$) faire
  - $P_r = P_r - 1$ et $RplusW = RplusW + 1$.
  - Ordonnancer *op* sur un port mémoire en lecture au cycle *C* et retourner la date de disponibilité de ce port sous forme <#P_r, disponible_quand>.
  - Mettre <#P_r, disponible_quand> dans la liste des dates de disponibilité des ressources.
  - Supprimer *op* de $Boite_{temp}$ et de la liste S_{P_r}.

  /* Ordonnancement des opérations *op* de $Boite_{temp}$ sur un port mémoire en écriture.*/
  Tant que ($Boite_{temp} \neq \varnothing$ et $op \in (Boite_{temp} \cap S_{P_w})$ et $P_w \neq 0$ et $RplusW < Borne$) faire
  - $P_w = P_w - 1$ et $RplusW = RplusW + 1$.
  - Ordonnancer *op* sur un port mémoire en écriture au cycle *C* et retourner la date de disponibilité de ce port sous forme <#P_w, disponible_quand>.
  - Mettre <#P_w, disponible_quand> dans la liste des dates de disponibilité des ressources.
  - Supprimer *op* de $Boite_{temp}$ et de la liste S_{P_w}.
  - S'il y a encore des noeuds non ordonnancés, ramasser les ressources disponibles au cycle *C*+1 à partir de la liste de disponibilité des ressources. Brancher au début de la boucle.

/* Calcul d'une borne inférieure sur le temps d'exécution du BB. */

- Retourner $Binf = Max_{o_i}\{T_{o_i} - ALAP_{o_i}\} + CC$, où T_{o_i} est le cycle d'ordonnancement de o_i.

Fin.

Figure 2.5 Adaptation de l'algorithme de calcul d'une borne inférieure sur le temps d'exécution d'un BB.

2.3.3- Borne supérieure sur le temps d'exécution d'un bloc de base

Le poids de chaque noeud (opération atomique) du graphe correspondant à un bloc de base est égal au temps nécessaire à l'exécution de cette opération. Une borne supérieure sur le temps d'exécution, *Bsup*, d'un bloc de base (BB) est le temps nécessaire pour l'exécution de tous ses noeuds. Nous pouvons donc obtenir une borne *Bsup* d'un BB en ordonnançant les noeuds de celui-ci sur l'architecture du système pour lequel nous voulons estimer les performances. Pour cet ordonnancement, nous considérons la contrainte des dépendances, la contrainte de ressources et la contrainte des interconnexions. Puisque nous nous intéressons à un modèle d'architecture à bus (voir Section 2.1), nous proposons un algorithme vorace pour l'ordonnancement des noeuds d'un BB comme il est montré à la Figure 2.6. Cet algorithme est une version modifié de l'algorithme vorace reconnu dans la littérature sous le nom "List-Scheduling" [DeMi94] et [GonG94]. Nous avons implémenté cet algorithme dans l'outil *C_PerfEstim.* Nous présenterons cet outil au Chapitre 5.

Dans l'algorithme présenté à la Figure 2.6, nous considérons la contrainte des dépendances, la contrainte des ressources et la contrainte des interconnexions. L'idée de base dans cet algorithme est la suivante. Tant que les noeuds du bloc de base BB ne sont pas tous ordonnancés, l'algorithme procède comme suit pour chaque cycle. Premièrement, on collecte les noeuds qui n'ont pas de prédécesseur non-ordonnancé. Par la suite, on distribue chaque noeud sur toutes les listes des ressources que celui-ci peut requérir. Deuxièment, on ordonnance temporairement les noeuds collectés pour le cas des unités fonctionnelles, et le cas des ports en lecture et des ports en écriture pour la mémoire. Un noeud est dit temporairement ordonnancé s'il est affecté à une ressource fonctionnelle ou à un port mémoire, mais il attend la disponibilité d'un bus pour être définitivement ordonnancé. Troisièment, on ordonnance les noeuds qui sont dans un état temporairement ordonnancés si les bus qu'ils requièrent sont disponibles; autrement ces noeuds seront ajoutés aux noeuds temporairement ordonnancés du prochain cycle. Quatrièment, on recollecte les ressources qui seront libérées dans le cycle suivant. Finalement, on recommence à partir de la première étape s'il y a encore des noeuds non ordonnancés.

Afin de faciliter la compréhension de l'algorithme présenté à la Figure 2.6, nous donnons les définitions et les notations suivantes:

- $M_{o_i} = ALAP_{o_i} - ASAP_{o_i}$ est la mobilité du noeud o_i où $ALAP_{o_i}$ et $ASAP_{o_i}$ sont les dates d'ordonnancement de o_i qui sont données, respectivement, par les algorithmes *ALAP* et *ASAP*. Dans notre algorithme, nous nous servons de la mobilité comme fonction de priorité quand nous voulons définir une liste avec priorité. La mobilité de chaque noeud est calculée une fois pour toute dans les premières étapes de cet algorithme. D'autres fonctions de priorité existent, mais la mobilité est la fonction la plus conseillée dans la littérature à ce sujet [GajD92]. Donc, dans une liste avec priorité, un élément est prioritaire si sa mobilité est plus petite que celles des autres.
- Pour les unités fonctionnelles
  - k est le nombre de types différents d'unités fonctionnelles.
  - Q_{T_j} est la liste des unités fonctionnelles de type T_j. $j = 1, \ldots, k$.
  - L_{T_j} est la liste, avec priorité, des noeuds prêts à ordonnancer sur une unité fonctionnelle de type T_j. $j = 1, \ldots, k$.
- Pour la mémoire
  - P_r est le nombre de ports en lecture. Certains de ces ports peuvent être en écriture.
  - P_w est le nombre de ports en écriture. Certains de ces ports peuvent être en lecture.
  - L_{P_r} est la liste, avec priorité, des noeuds prêts à ordonnancer sur un port en lecture.
  - L_{P_w} est la liste, avec priorité, des noeuds prêts à ordonnancer sur un port en écriture.
  - *Borne* = $(P_r + P_w)$ - le nombre de ports en lecture et en écriture. *Borne* est le nombre total maximal des lectures de la mémoire et des écritures dans celle-ci qu'on peut faire pendant un cycle donné.
  - *RplusW* est le nombre total des lectures de la mémoire et des écritures dans celle-ci pendant un cycle donné. Ce nombre ne doit pas dépasser *Borne*.

- $L_{ops\_prêtes}$ est la liste des noeuds prêts à ordonnancer pendant un cycle donné.
- L_{attend_bus} est la liste des couples <R_{T_j}, *op*>, où R_{T_j} désigne une ressource de type T_j et *op* désigne une opération. Ceci signifie que *op* est temporairement ordonnancé sur R_{T_j} mais il attend la disponibilité d'un bus pour être définitivement ordonnancé. L_{attend_bus} est avec priorité définie à partir de la mobilité des opérations *op* de ses éléments <ressource R_{T_j}, opération *op*>.

Entrées: Un bloc de base (*BB*) et la description de l'architecture d'un système.
Sortie: Une borne supérieure (*Bsup*) sur le temps d'exécution du bloc de base en entrée.
Début

- Insérer les unités fonctionnelles de type T_j dans la liste Q_{T_j}. $j = 1, ..., k$.
- Pour la mémoire, initialiser le nombre de ports en lecture P_r et le nombre de ports en écriture P_w.
- Faire: $Borne = (P_r + P_w)$ - le nombre de ports mémoire en lecture et en écriture.
- Trouver le $ASAP_{o_i}$ et le $ALAP_{o_i}$ pour chaque noeud o_i du *BB*.
- Calculer la mobilité M_{o_i} pour chaque noeud o_i (i.e., $M_{o_i} = ALAP_{o_i} - ASAP_{o_i}$).
- Initialiser *Bsup* et le compteur du temps *C* à 0.
- Tant que (tous les noeuds ne sont pas ordonnancés) faire
  - $C = C + 1$, $RplusW = 0$ et $L_{ops\_prêtes} = \emptyset$.
  - Insérer les noeuds prêts à ordonnancer au cycle *C* dans la liste $L_{ops\_prêtes}$.
  - Distribuer les éléments *de* $L_{ops\_prêtes}$ sur les listes L_{T_i}, L_{P_r} et L_{P_w} qui leur correspondent.

  /* Assignation d'une unité fonctionnelle de type T_j.*/
  Pour $j = 1$ à k faire
  Tant que ($L_{T_j} \neq \emptyset$ et $Q_{T_j} \neq \emptyset$) faire
  - Supprimer une unité fonctionnelle R_{T_j} de Q_{T_j}.
  - Supprimer un élément op_i de L_{T_j}. Supprimer op_i des autres listes.
  - Insérer $< R_{T_j}, op_i >$ dans la liste L_{attend_bus}.

  /* Assignation d'un port mémoire en lecture.*/
  Tant que ($L_{P_r} \neq \emptyset$ et $P_r \neq 0$ et $RplusW < Borne$) faire
  - $P_r = P_r - 1$ (i.e., on occupe un port P_{ri}).
  - Supprimer un élément op_i de L_{P_r}.
  - Insérer $<P_{ri}, op_i>$ dans la liste L_{attend_bus}.
  - $RplusW = RplusW + 1$.

  /* Assignation d'un port mémoire en écriture.*/
  Tant que ($L_{P_w} \neq \emptyset$ et $P_w \neq 0$ et $RplusW < Borne$) faire
  - $P_w = P_w - 1$ (i.e., on occupe un port P_{wi}).
  - Supprimer un élément op_i de L_{P_w}.
  - Insérer $<P_{wi}, op_i>$ dans la liste L_{attend_bus}.
  - $RplusW = RplusW + 1$.

  /* Assignation des bus.*/
  Tant que (il y a des bus disponibles et $L_{attend\_bus} \neq \emptyset$) faire
  - Supprimer un élément <ressource de type *T*, op_i> de L_{attend_bus}.
  - Si le bus que op_i requiert est disponible alors
    - Ordonnancer op_i sur la ressource de type *T* au cycle *C* et retourner la date de la prochaine disponibilité de celle-ci.
    - $Bsup = \max(Bsup, (C - 1) +$ la latence de $op_i)$.
    - Insérer <ressour. de type *T*, disponible quand> dans la liste de disponibilité des res.
    - Modifier les successeurs de op_i en modifiant la date pour laquelle ils peuvent être prêts à ordonnancer.
  - Sinon insérer <ressource de type *T*, op_i> dans une liste temporaire *Temp*.
  - Si $Temp \neq \emptyset$ insérer les éléments de *Temp* dans L_{attend_bus} et mettre $Temp = \emptyset$.
  - Récupérer les bus disponibles au cycle *C*+1.
  - Récupérer les ressources disponibles au cycle *C*+1 et les distribuer sur les listes qui leur correspondent. Aller au début de la boucle la plus externe.
- Return (*Bsup*).

Fin.

Figure 2.6 Adaptation de l'algorithme de calcul d'une borne supérieure sur le temps d'exécution d'un BB.

2.3.4- Borne inférieure sur le temps d'exécution d'un GCFD

Une borne inférieure sur le temps d'exécution d'un graphe de contrôle et de flot de données (GCFD) peut être calculée en déterminant la longueur de plus court chemin dans le graphe GCFD. Pour cela, on suppose que le poids de chaque noeud (bloc de base BB) du graphe de flot de contrôle (GFC) est égal à la valeur de la borne inférieure sur son temps d'exécution; dans ce GFC, l'exécution des blocs de base est séquentielle et le poids de chaque arc est égal au poids de son noeud origine. Cette méthode s'applique directement quand le GFC ne contient pas des boucles. Pour calculer la longueur de plus court chemin dans un GFC dans le cas où il y a des boucles, une idée est de dérouler complètement ces boucles et ensuite de calculer la longueur de plus court chemin dans le GFC résultant. Cette idée ne fonctionne pas en général, car la taille du code résultat peut être extrêmement grande. Une solution à ce problème a été proposée par Bennour et *al.* [BenL96]. Ces auteurs supposent dans ce cas que les boucles itèrent un nombre minimal de fois. Sous cette considération, leur algorithme (voir Figure 2.7) commence par l'enlèvement des boucles dans le graphe GFC en commençant par la boucle la plus interne. À chaque fois qu'il rencontre le début d'une boucle, l'algorithme calcule la valeur V de la longueur de plus court chemin dans le corps de celle-ci. Ensuite, le corps de la boucle est remplacé par un seul noeud avec un poids égal au résultat de la multiplication de la valeur V par le nombre minimal d'itérations de la boucle. Une fois toutes les boucles du graphe GFC remplacées, l'algorithme calcule la valeur de la longueur de plus court chemin dans le graphe résultant. Cette valeur est donc une borne inférieure sur le temps d'exécution du graphe GCFD.

2.3.5- Borne supérieure sur le temps d'exécution d'un GCFD

Dans le cas où le graphe de flot de contrôle (GFC) ne contient pas de boucles, le calcul d'une borne supérieure sur le temps d'exécution du graphe de contrôle et de flot de données (GCFD) consiste à calculer la valeur de la longueur de plus long chemin dans le graphe GFC en supposant que le poids de chaque noeud (bloc de base BB) est égal à une borne supérieure sur son temps d'exécution; dans ce graphe, le poids de chaque arc est égal au poids de son noeud origine. Quand le graphe GFC contient des boucles, Bennour et *al.* [BenL96] ont proposé l'algorithme présenté à la Figure 2.8. L'idée de base dans cet algorithme est la même que dans le cas de calcul d'une borne inférieure sur le temps d'exécution d'un graphe GCFD présentée à la Section 2.3.4.

Entrées: Un graphe de contrôle et de flot de données (GCFD), une borne inférieure sur le temps d'exécution de chaque bloc de base (BB) du graphe GCFD ainsi que le nombre minimal d'itérations de chacune de ses boucles.

Sortie: Une borne inférieure sur le temps d'exécution de celui-ci.

Début

- Associer à chaque noeud du graphe du flot de contrôle (GFC) un poids égal à la borne inférieure sur le temps d'exécution du BB correspondant.
  - Tant que (il y a des boucles dans le graphe GFC) faire
    - Pour chaque boucle la plus interne faire
      - Calculer la valeur *V* de la longueur de plus court chemin dans son corps.
      - Remplacer son corps par un seul noeud.
      - Associer à ce noeud un poids égal à (*V**nombre min. d'itérations de celle-ci).
- Calculer la valeur de la longueur de plus court chemin dans le graphe GFC résultat.
- Retourner cette valeur comme borne inférieure sur le temps d'exécution du graphe GCFD.

Fin.

Figure 2.7 Algorithme pour le calcul d'une borne inférieure sur le temps d'exécution d'un GCFD.

Entrées: Un graphe de contrôle et de flot de données (GCFD), une borne supérieure sur le temps d'exécution de chaque bloc de base (BB) du graphe GCFD ainsi que le nombre maximal d'itérations de chacune de ses boucles.

Sortie: Une borne supérieure sur le temps d'exécution du graphe GCFD.

Début

- Associer à chaque noeud du graphe du flot de contrôle (GFC) un poids égal à la borne supérieure sur le temps d'exécution du BB correspondant.
  - Tant que (il y a des boucles dans le graphe GFC) faire
    - Pour chaque boucle la plus interne faire
      - Calculer la valeur *V* de la longueur de plus long chemin dans son corps.
      - Remplacer son corps par un seul noeud.
      - Associer à ce noeud un poids égal à (*V**nombre max. d'itérations de celle-ci).
- Calculer la valeur de la longueur de plus long chemin dans le graphe GFC résultat.
- Retourner cette valeur comme borne supérieure sur le temps d'exécution du graphe GCFD.

Fin.

Figure 2.8 Algorithme pour le calcul d'une borne supérieure sur le temps d'exécution d'un GCFD.

Chapitre 3

Modification du compilateur *LCC*

Les algorithmes d'estimation des performances d'un système, vus au chapitre précédent, ont besoin d'un graphe de contrôle et de flot de données (GCFD). Nous rappelons (voir Chapitre 2) qu'un GCFD est une représentation intermédiaire d'un programme donné. Pour traduire automatiquement un programme en un GCFD, le besoin d'un compilateur s'impose.

Pour remédier à ce besoin, la conception d'un compilateur ou l'utilisation d'un compilateur existant constitue une solution. La conception d'un compilateur est une tâche coûteuse en terme de temps. Par conséquent, quand on s'intéresse à une estimation rapide des performances d'un système, il nous reste la possibilité d'utiliser un compilateur reciblable existant, tel que le compilateur *LCC*.

LCC "Local C Compiler" est un compilateur séquentiel pour ANSI C. Il a été développé à l'Université Princeton et aux laboratoires AT&T de Bell. Il se caractérise par sa rapidité d'exécution et par une petite taille par rapport aux autres compilateurs existants ([FraH91a], [FraH91b] et [FraH95]). Des versions du compilateur *LCC* pour le SPARC, le MIPS, le X86, le Motorola 68020 et le VAX sont disponibles dans le domaine public [LCCV3.5]. La version 3.5 du compilateur *LCC* se compose des interfaces suivantes: *symbolic*, *sparc-sun*, *sparc-solaris*, *x86-dos*, *mips-irix*, *mips-ultrix*, *null* et *NULL*. Son interface *symbolic* génère, pour un code en Langage C, un graphe de contrôle et de flot de données (GCFD).

Tel que généré par le compilateur *LCC*, le GCFD ne répond pas aux besoins des algorithmes d'estimation de performances vus au chapitre précédent. Afin de répondre à ces besoins, les deux problèmes suivants doivent être résolus: l'ajout de dépendances supplémentaires dans ce GCFD et, la détermination du début et de la fin de chaque boucle ainsi que de ses nombres minimal et maximal d'itérations quand ce graphe correspond à un traitement itératif.

Dans ce chapitre, nous nous concentrons sur la résolution du problème de dépendances dans le graphe généré par le compilateur *LCC* et sur la reconstruction d'un nouveau GCFD. Pour ce GCFD, nous résolvons le problème de la détection des boucles et des nombres minimal et maximal d'itérations de chacune de celles-ci. Afin de visualiser graphiquement ce GCFD, nous développons une interface permettant de communiquer ce compilateur avec un outil de visualisation graphique d'un graphe appelé "Visualization of Compiler Graphs" (*VCG*).

Le reste de ce chapitre est organisé comme suit. La première section présente le compilateur *LCC*. Dans la deuxième section, nous traitons les modifications et les extensions que nous avons apportées au compilateur *LCC*. À la troisième section, nous examinons l'intégration de l'outil *VCG* dans le compilateur *LCC*. Finalement, nous donnons nos conclusions dans la quatrième section.

3.1- Le compilateur *LCC*

Cette section se compose des sous-sections suivantes. Dans la première sous-section, nous donnons une vue d'ensemble du compilateur *LCC*. La deuxième sous-section présente un exemple de graphe généré par le compilateur *LCC*. À la troisième sous-section, nous examinons la traduction des boucles *Do*, *while* et *For* par le compilateur *LCC*.

3.1.1- Vue d'ensemble du compilateur *LCC*

Le compilateur *LCC* se compose d'une partie dite "target-independent" aussi appelée frontal et d'une partie dite "target-dependent" aussi appelé dorsal. Les mots frontal et dorsal sont, respectivement, la traduction des mots anglais "front end" et "back end". Le frontal et le dorsal communiquent entre eux à l'aide d'une interface. Cette interface est constituée de quelques structures de données, partagées par le frontal et le dorsal, de 18 fonctions (voir Tableau 3.1) et de 36 opérateurs (voir Tableau 3.2).

Le frontal appelle le dorsal pour générer et émettre le code. Le dorsal a besoin du frontal pour la gestion des "Directed Acyclic Graphs" (DAGs), des symboles et des chaînes de caractères, et pour le perfectionnement de la sortie. Pour plus de détails à ce sujet, le lecteur peut se référer à

[FraH95] et à [FraH91b].

Comme nous l'avons mentionné au début de ce chapitre, la version 3.5 du compilateur *LCC* se compose des interfaces suivantes: *sparc-sun*, *sparc-solaris*, *symbolic*, *x86-dos*, *mips-irix*, *mips-ultrix*, *null* et *NULL*. Par exemple, l'interface *sparc-solaris* génère le code pour les machines sparc qui ont le système d'exploitation solaris et l'interface *symbolic* génère un GCFD. Ces interfaces sont définies par la structure de données suivante:

```
typedef struct binding{
        char *name;
        Interface *ir;
}Binding;
```

Dans l'interface *symbolic*, la fonction *gen* reçoit un ensemble de DAGs de la fonction *gencode* du frontal. Ensuite, *gen* linéarise chaque DAG dans l'ordre de son exécution, affecte un numéro à chacun de ses noeuds et retourne un pointeur sur celui-ci à la fonction *emitcode* du frontal. Par la suite, *emitcode* passe ce pointeur à la fonction *emit*. Finalement, *emit* affiche les noeuds du DAG référencé par le pointeur qu'il a reçu. La fonction *progend* ne fait rien; son corps est vide. Nous exploitons cette fonction pour l'appel de la majorité des fonctions que nous avons ajoutées au compilateur *LCC* comme il est montré dans la Figure 3.6 plus loin.

```
void address (Symbol p, Symbol q, int n)
void blockbeg (Env *e)
void blockend (Env *e)
void defaddress (Symbol p)
void defconst (int ty, Value v)
void defstring (int len, char *s)
void defsymbol (Symbol p)
void emit (Node p)
void export (Symbol p)
void function (Symbol f, Symbol caller[], Symbol callee[], int ncalls)
Node gen (Node p)
void global (Symbol p)
void import (Symbol p)
void local (Symbol p)
void progbeg (int argc, char *argv[])
void progend (void)
void segment (int s)
void space (int n)
```

Tableau 3.1 Fonctions de l'interface entre le frontal et le dorsal.

Opérateur	Opération
ADDRF	adresse d'un paramètre
ADDRG	adresse d'une variable globale
ADDRL	adresse d'une variable locale
CNST	constante
BCOM	complément à un
CVC	conversion à partir d'un char
CVD	conversion à partir d'un double
CVF	conversion à partir d'un float
CVI	conversion à partir d'un int
CVP	conversion à partir d'un pointeur
CVS	conversion à partir d'un short
CVU	conversion à partir d'un unsigned
INDIR	indirection
NEG	négation
ADD	addition
BAND	and bit à bit
BOR	or bit à bit
BXOR	xor bit à bit

Opérateur	Opération
DIV	division
LSH	décalage à gauche
MOD	modulo
MUL	multiplication
RSH	décalage à droite
SUB	soustraction
ASGN	assignation à une adresse
EQ	brancher si c'est égal
GE	brancher si c'est supér. ou ég.
GT	brancher si c'est supérieur
LE	brancher si c'est inf. ou égal
LT	brancher si c'est inférieur
NE	brancher si ce n'est pas égal
ARG	argument
CALL	appel d'une fonction
RET	retourner à partir d'une fonct.
JUMP	brancher
LABEL	définition d'une étiquette

Tableau 3.2 L'ensemble des opérateurs de l'interface entre le frontal et le dorsal.

3.1.1- Exemple de graphe généré par la version originale du compilateur *LCC*

Soit le code en Langage C suivant:

```
main(){
      int a, b;

      b = a + 1;
      a = 2;
}
```

Pour ce code, l'utilisation de l'interface *symbolic* du compilateur *LCC* génère le GCFD présenté à la Figure 3.1. L'affichage de ce graphe est sous un format textuel. Ce format, adéquat pour un traitement machine, rend difficile la compréhension par un humain. Pour y remédier, nous intégrons un outil de visualisation graphique d'un graphe appelé *VCG* dans le compilateur *LCC*. Nous revenons à la discussion de l'intégration de l'outil *VCG* dans le compilateur *LCC* à la Section 3.3.

Par exemple, en examinant les lignes suivants de la Figure 3.1:

```
node#2 ADDRLP count=1 b
node#5 ADDRLP count=1 a
node#4 INDIRI count=1 #5
```

La première ligne signifie que le noeud #2 correspond à l'adresse de `b` et que celui-ci est référé une seule fois (i.e., signification de `count=1`) par les autres noeuds du même "Directed Acyclic Graph" (DAG). En changeant `b` par `a`, la deuxième ligne a une même signification que celle de la première ligne. La troisième ligne signifie que le noeud #4 correspond au contenu de l'adresse représentée par le noeud #5, et que celui-ci est référé une seule fois par les autres noeuds du même DAG.

La représentation graphique du graphe de la Figure 3.1 est donnée à la Figure 3.2. Dans celle-ci, les arcs représentent les dépendances de données entre les noeuds de ce graphe. Les opérateurs correspondant aux noeuds 4 et 7 sont, respectivement, un opérateur de lecture et un opérateur d'écriture. Ces deux opérateurs accèdent une même adresse mémoire (i.e., adresse de a). Cette adresse est représentée par les noeuds 5 et 8. Une exécution correcte de ce graphe est donnée par son exécution séquentielle. Dans ce cas, le noeud 4 s'exécute toujours avant le noeud 7. Donc, la lecture du contenu de l'adresse de `a` par le noeud 4 se fait avant la modification du contenu de cette adresse par le noeud 7. Pour exploiter ce graphe pour des fins de parallélisme, un arc du noeud 4 vers le noeud 7 doit être ajouté comme il est illustré par la flèche pointillée de la Figure 3.2. Cet arc permet de bloquer l'exécution du noeud 7 jusqu'à ce que le noeud 4 soit exécuté. Autrement, le noeud 7 peut être exécuté avant le noeud 4 ce qui est contradictoire avec le cas d'une exécution séquentielle. Un de nos objectifs dans le reste de ce chapitre est la résolution de ce genre de situations comme nous allons voir à la Section 3.2.

```
export main
segment text
function main type=int function(void) class=auto scope=GLOBAL ref=0
local a type=int class=auto scope=LOCAL offset=0 ref=2000
local b type=int class=auto scope=LOCAL offset=4 ref=1000
maxoffset=8

node#2 ADDRLP count=1 b
node#5 ADDRLP count=1 a
node#4 INDIRI count=1 #5
node#6 CNSTI count=1 1
node#3 ADDI count=1 #4 #6
node'1 ASGNI count=0 #2 #3 4 4
node#8 ADDRLP count=1 a
node#9 CNSTI count=1 2
node'7 ASGNI count=0 #8 #9 4 4

1:
end main
```

Figure 3.1 Exemple de graphe généré par la version originale du compilateur LCC.

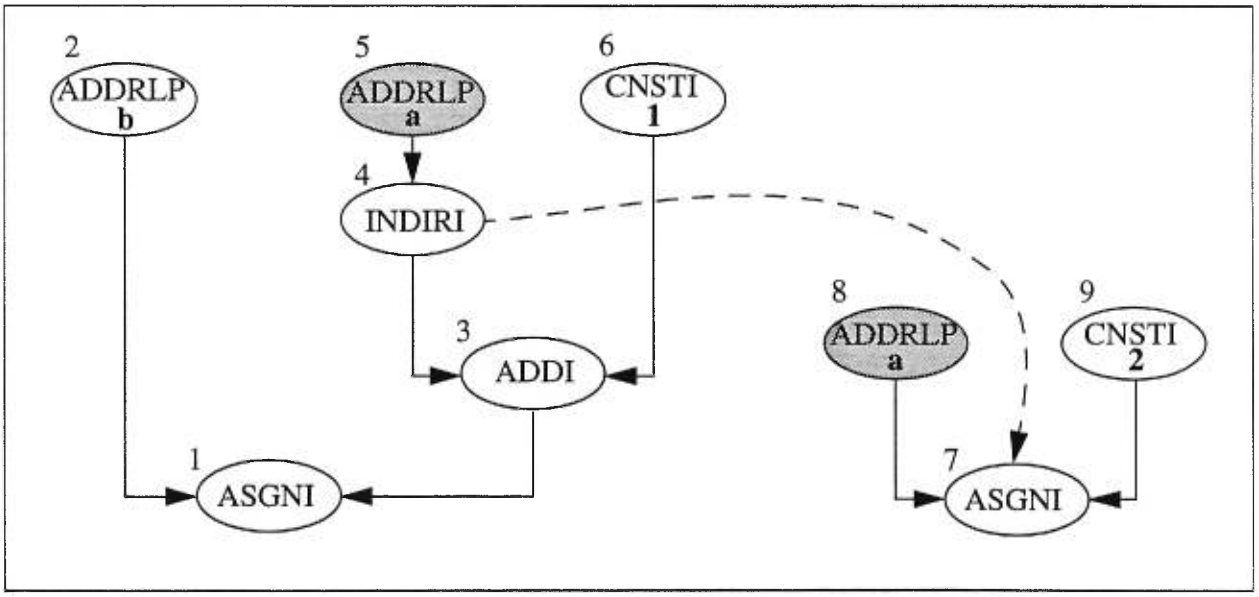


Figure 3.2 Affichage graphique du graphe de la Figure 3.1.

3.1.2- Transformation des boucles par le compilateur *LCC*

Les boucles considérées dans ce chapitre sont la boucle *Do*, la boucle *While* et la boucle *For*. Ces boucles sont celles définies dans le langage C selon le standard ANSI. Pendant la phase d'analyse sémantique, le compilateur *LCC* transforme ces boucles en un format utilisant des étiquettes et aller-à. Dans le reste de cette section, nous présentons la transformation des boucles *Do*, *While* et *For* par le compilateur *LCC*.

3.1.2.1- Transformation de la boucle *Do*

En utilisant des étiquettes, le compilateur *LCC* transforme cette boucle sous la forme:

L:
corps de la boucle
L+1:
test sur la condition de retour à l'étiquette L

3.1.2.2- Transformation de la boucle *While*

En utilisant des étiquettes, le compilateur *LCC* transforme cette boucle sous la forme:

brancher à L+1
L:
corps de la boucle
L+1:
test sur la condition de retour à l'étiquette L
L+2:

3.1.2.3- Transformation de la boucle *For*

La forme générale de cette boucle est: *for* $(e_1; e_2; e_3)$ *{corps}* où $e_{i = 1, 2, 3}$ sont des expressions. Par exemple, pour le code suivant: `for (i=0; i<100; i++) {;}`, les expressions $e_{i = 1, 2, 3}$ valent, respectivement, `i=0`, `i<100` et `i++`. La transformation de cette boucle par le compilateur *LCC* dépend de l'existence des $e_{i = 1, 2, 3}$ et du nombre d'exécutions du corps de celle-ci. Par exemple, quand toutes les expressions $e_{i = 1, 2, 3}$ sont présentes et qu'on n'est

pas sûr que le corps de la boucle sera exécuté au moins une fois, le compilateur *LCC* donne la transformation suivante:

e_1
brancher à L+3
L:
corps de la boucle
L+1:
e_3
L+3:
si e_2 est vraie brancher à l'étiquette L
L+2:

3.2- Modification du compilateur *LCC*

Comme nous l'avons mentionné au début de ce chapitre, les objectifs de la modification du compilateur *LCC* sont: la résolution du problème de dépendances dans le graphe de contrôle et de flot de données (GCFD) généré par *LCC*, la détection de boucles, la détermination du nombre minimal et du nombre maximal d'itérations de chacune de celles-ci, ainsi que l'intégration d'un outil de visualisation graphique d'un graphe dans le compilateur *LCC*. Cette intégration sera examinée à la Section 3.3.

Dans les sous-sections qui suivent, nous revenons plus en détail sur le problème de dépendances dans le graphe généré par le compilateur *LCC*, sur le problème de la localisation de boucles et la détermination des nombres minimal et maximal d'itérations de chacune de celles-ci, ainsi que sur la description des modifications apportées au compilateur *LCC*.

3.2.1- Le problème de dépendances dans le graphe de contrôle et de flot de données généré par le compilateur *LCC*

Soient N_1 et N_2 deux noeuds d'un bloc de base (BB). Si N_1 et N_2 accèdent une même adresse mémoire aux instants t_1 et t_2, si aucun autre noeud n'accède celle-ci entre ces deux instants, et si au moins un de ces deux noeuds est une opération d'écriture dans cette adresse, alors une

dépendance entre N_1 et N_2 existe. Si pour une exécution séquentielle du BB, l'exécution de N_1 se fait avant celle de N_2 alors pour exploiter ce BB, pour des fins de parallélisme, un arc de N_1 vers N_2 doit être ajouté si ce dernier n'existe pas déjà. Cet arc permet de bloquer l'exécution de N_2 jusqu'à ce que N_1 soit exécuté; sinon N_2 peut être exécuté avant N_1, ce qui est contradictoire avec l'exécution séquentielle. Notons que l'exécution séquentielle contribue à fixer la sémantique de l'exécution de ce BB.

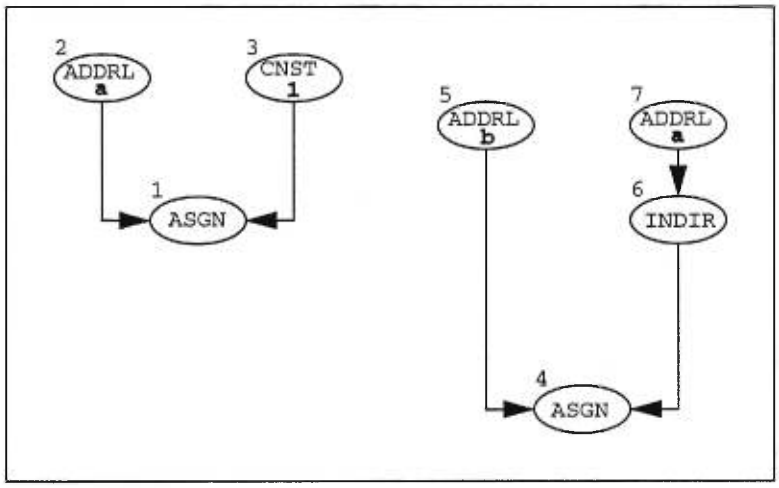
Quand N_1 et N_2 appartiennent à un même bloc de base, la dépendance entre N_1 et N_2 est dite de type intra-bloc. Autrement, cette dernière est dite de type inter-blocs.

Dans les blocs de base du GCFD généré par le compilateur *LCC*, les valeurs possibles pour la paire (N_1, N_2) qui impliquent des dépendances de données implicites sont (INDIR, ASGN), (ASGN, INDIR) et (ASGN, ASGN) où INDIR est un opérateur de lecture et ASGN est un opérateur d'écriture. Quand la paire (N_1, N_2) vaut (INDIR, ASGN), on dit que nous avons une dépendance de données de type anti-dépendance. Nous disons que nous avons une dépendance de flot si cette paire vaut (ASGN, INDIR). Finalement, on dit que nous avons une dépendance de sortie si (N_1, N_2) vaut (ASGN, ASGN).

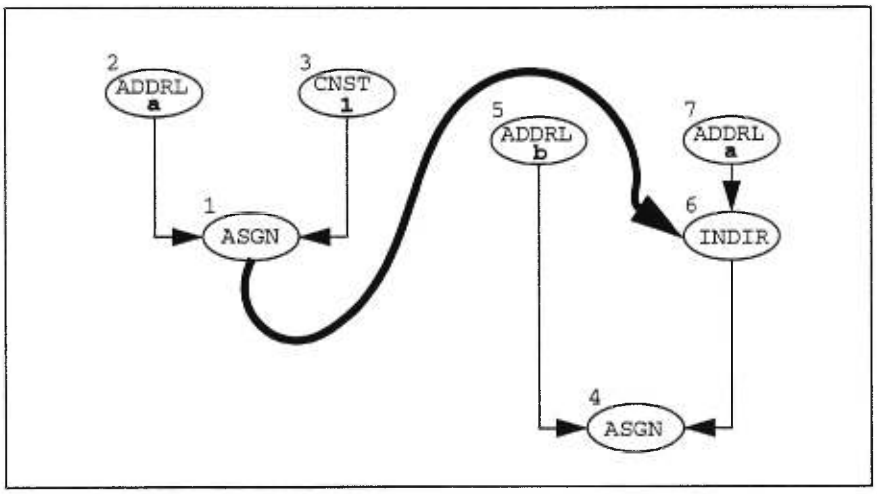
Le GCFD généré par la version originale du compilateur *LCC* est destiné à une exécution séquentielle de celui-ci. Pour rendre ses blocs de base utilisables en vue d'une exécution parallèle, nous avons résolu le problème des dépendances de type intra-bloc; la résolution du problème des dépendances inter-blocs n'est pas l'objet de ce chapitre, car l'ordre d'exécution de ces blocs de base n'a pas d'impacts sur les résultats de ce travail. Nous avons réalisé cette résolution à l'aide de la fonction *emit1* comme nous allons voir plus loin. Les Figures 3.3 et 3.4 montrent les graphes générés avant et après la modification du compilateur *LCC*. Ces figures correspondent aux dépendances de type dépendance de flot et de type dépendance de sortie. Dans ces figures, les arcs de dépendance ajoutés après la modification du compilateur *LCC* sont affichés en gras. Le problème de dépendance de type anti-dépendance est examiné à la Section 3.1.1.

```
main()
  {
    int a, b;
    a = 1;
    b = a;
  }
```

(a) Code en Langage C.

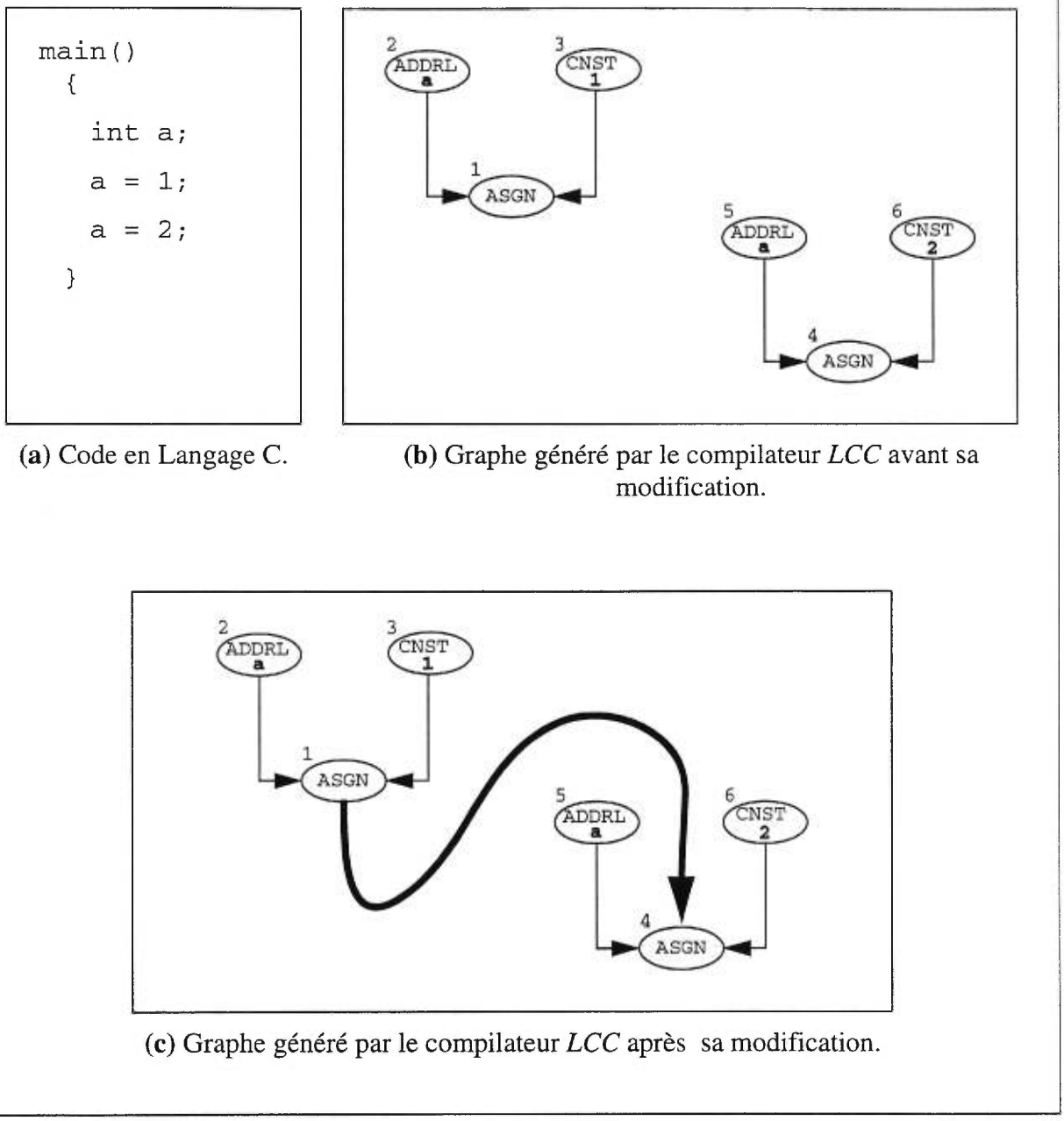


(b) Graphe généré par le compilateur *LCC* avant sa modification.



(c) Graphe généré par le compilateur *LCC* après sa modification.

Figure 3.3 Dépendance de type dépendance de flot.



(a) Code en Langage C.

(b) Graphe généré par le compilateur *LCC* avant sa modification.

(c) Graphe généré par le compilateur *LCC* après sa modification.

Figure 3.4 Dépendance de type dépendance de sortie.

3.2.2- Localisation de boucles dans le graphe de contrôle et de flot de données généré par le compilateur *LCC*

Nous rappelons que les boucles sont transformées par le compilateur *LCC* en un format

avec des étiquettes (voir Section 3.1.2). Cette transformation ne permet pas de retrouver facilement ces boucles dans le graphe de contrôle et de flot de données (GCFD) généré par le compilateur *LCC*. Pour localiser les boucles dans un GCFD généré par le compilateur *LCC*, nous avons ajouté des structures de données et une fonction appelée *definelab_fin_boucl* au niveau du frontal. L'appel à cette fonction se fait comme il est montré dans la Figure 3.5. Les structures de données ajoutées permettent de garder les traces des boucles lors de la phase de l'analyse sémantique du compilateur *LCC*. Ces traces sont exploitées au niveau du dorsal pour déterminer, dans le GCFD, le bloc de base qui débute une boucle et le bloc de base qui la finit. La mise à jour de ces structures de données se fait par la fonction *definelab_fin_boucl*.

3.2.3- Détermination des nombres minimal et maximal d'itérations d'une boucle

Parmi nos objectifs dans la modification du compilateur *LCC*, on désire déterminer des nombres minimal et maximal d'itérations d'une boucle; même pour les cas des boucles *Do* et *While*. Dans la version originale du compilateur *LCC*, il n'y a pas de méthode pour déterminer ces nombres. Pour résoudre ce problème, nous avons proposé l'ajout des nombres minimal et maximal d'itérations de chaque boucle sous forme d'un commentaire spécial lors de l'utilisation de celle-ci. Ce commentaire est "*/*$ Min Max */*", où *Min* et *Max* désignent les nombres minimal et maximal d'itérations d'une boucle. Dans ce cas, on traite la séquence de caractères */*$* comme un mot clé réservé. L'utilisation de chaque boucle doit donc être comme suit (voir aussi un exemple dans la Figure 3.9 (a) plus loin):

```
Do /*$ Min  Max */ le reste de la boucle.
For (/*$ Min  Max */ le reste de la boucle.
While (/*$ Min  Max */ le reste de la boucle.
```

Le nombre *Min* et le nombre *Max* sont détectés par le compilateur *LCC* modifié dans la phase d'analyse lexicale. Ils sont stockés par la suite dans une structure de données FIFO ("First In First Out". Cette FIFO est exploitée dans la suite par le dorsal pour déterminer les nombres minimal et maximal d'itérations de chaque boucle. Notons que, dans ce cas, l'utilisation d'une FIFO suffit car l'ordre du traitement des boucles au niveau du frontal et au niveau du dorsal est

préservé.

3.2.4- Description des modifications apportées au compilateur *LCC*

Les modifications que nous avons apportées au compilateur *LCC* sont introduites dans son interface *symbolic*. Au lieu de changer les corps des fonctions de cette interface, nous avons ajouté un ensemble de structures de données et de fonctions. Les structures de données ajoutées sont en majorité dans un nouveau fichier appelé *struct.h*. Le reste de ces structures de données est dans le fichier *config.h* de la version originale du compilateur *LCC*. Ces structures de données gardent des traces provenant du frontal lors de la phase d'analyse lexicale et de la phase d'analyse sémantique. Ces traces sont utilisées dans la suite par le dorsal pour localiser les boucles et pour déterminer les nombres minimal et maximal d'itérations de celles-ci. L'appel aux fonctions que nous avons ajoutées se fait tel que montré aux Figures 3.5 et 3.6.

Nos modifications comportent sept étapes:

1) Construction d'un premier graphe temporaire G1 à partir de celui produit par la version originale du compilateur *LCC*. Pour cela, nous utilisons la fonction *inserer* (voir Figure 3.6).

2) Résolution du problème des dépendances de type intra-bloc (voir Section 3.2.1) pour les blocs de base du graphe G1 en construisant un deuxième graphe temporaire G2. Pour cette fin, nous utilisons la fonction *graphe_temp_2* (voir Figure 3.6).

3) Numérotation des blocs de base du graphe temporaire G2 et détermination des successeurs de chacun de ceux-ci. Pour cette fin, nous utilisons les fonctions *renumerote* et *dag_suiv* (voir Figure 3.6).

4) Détermination du bloc de base début et du bloc de base fin pour chaque boucle ainsi que le nombre minimal et le nombre maximal d'itérations de celle-ci. Pour cela, nous utilisons la fonction *boucl_iter* (voir Figure 3.6).

5) Fusionnement, si c'est possible, de certains blocs de base du graphe G2. Après ce

fusionnement, nous obtenons un troisième graphe temporaire G3. Nous réalisons cette étape à l'aide de la fonction *combine_bb* (voir Figure 3.6).

6) Perfectionnement du graphe G3 comme par exemple la détermination des successeurs et des prédécesseurs de chaque noeud de chacun de ses blocs, et l'ajout des noeuds de synchronisations pour chacun de ceux-ci. Après ce perfectionnement, nous obtenons le graphe de contrôle et de flot de données (GCFD) final. Nous réalisons cette étape à l'aide de la fonction *gcfd* (voir Figure 3.6).

7) Génération de la description du GCFD final sous le format reconnu par l'outil *VCG*. Pour cette fin, nous utilisons la fonction *vcg_gcfd* (voir Figure 3.6).

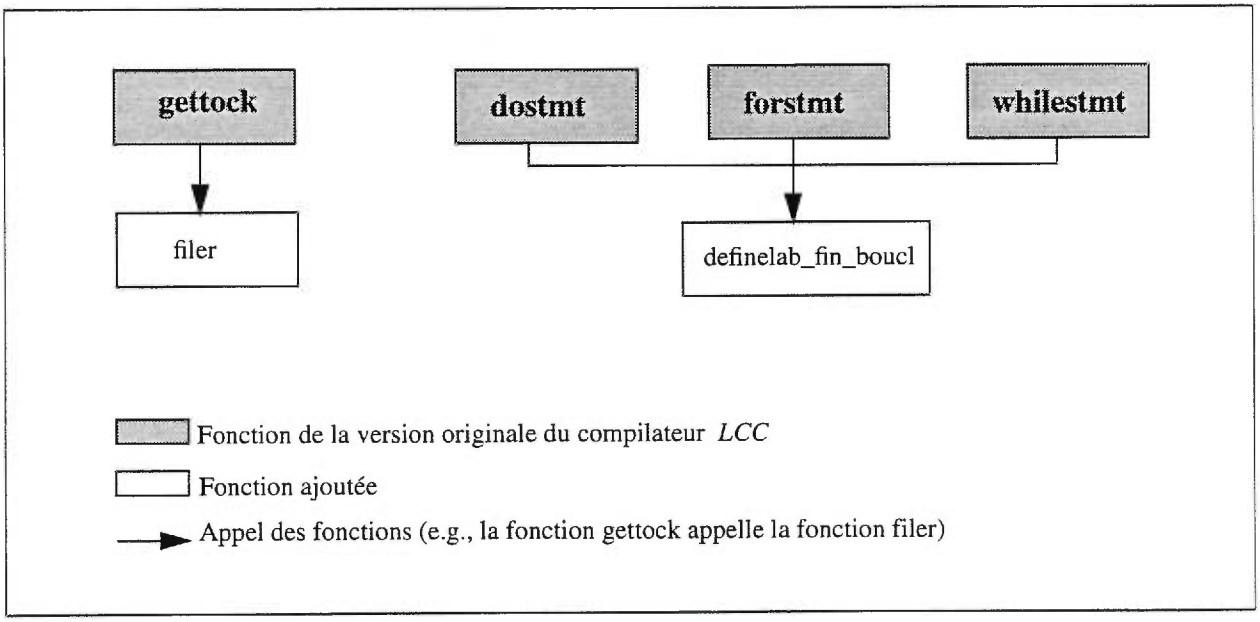


Figure 3.5 L'ensemble de fonctions ajoutées au niveau du frontal à l'interface symbolic du compilateur LCC.

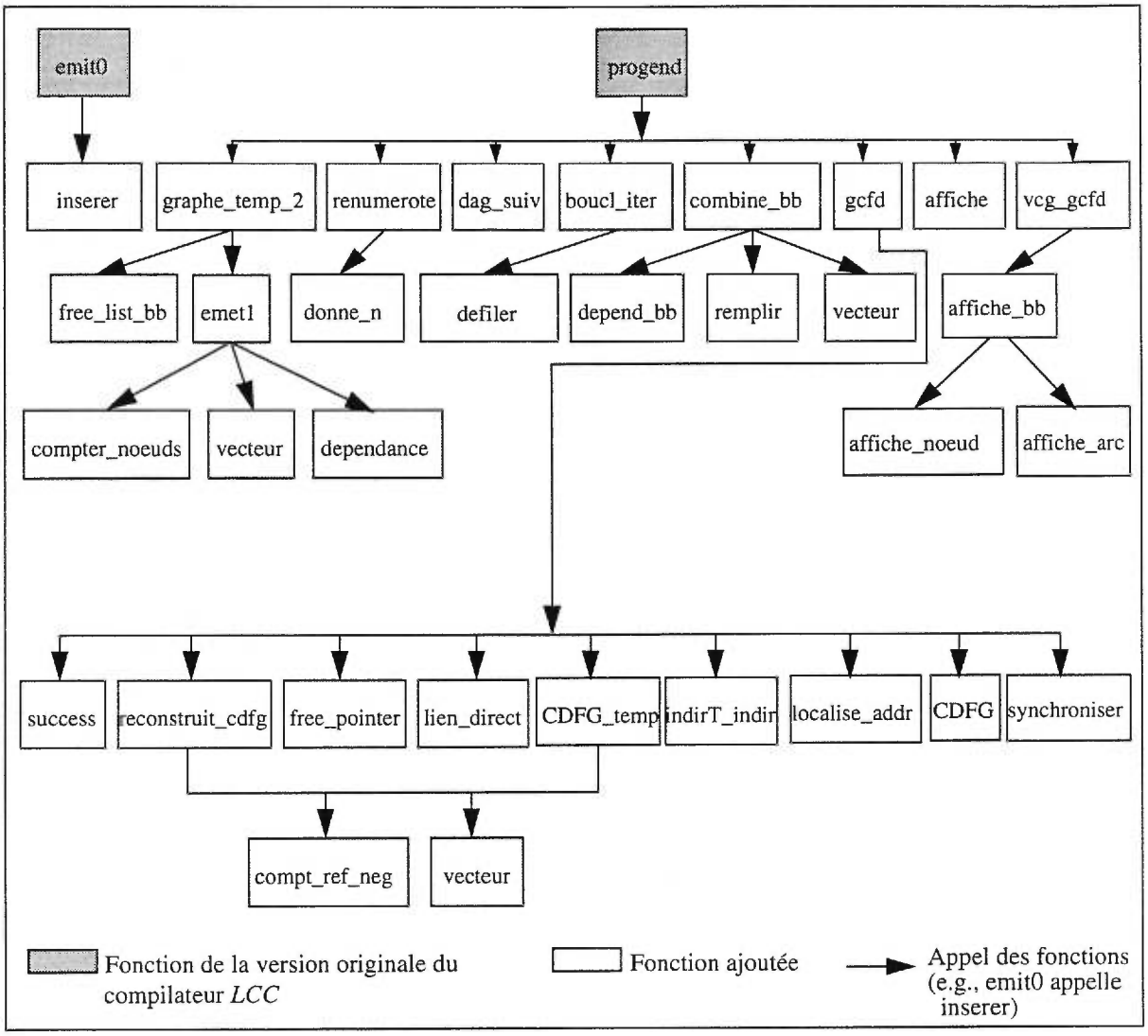


Figure 3.6 L'ensemble de fonctions ajoutées au niveau du dorsal à l'interface symbolic du compilateur LCC.

3.3- Intégration de l'outil *VCG* dans le compilateur *LCC* modifié

Comme nous l'avons vu à la Section 3.1.1, l'affichage du graphe de contrôle et de flot de données (GCFD) généré par l'interface *symbolic* de la version originale du compilateur *LCC* est sous un format textuel. Sous ce format, le GCFD généré par cette interface est difficile à comprendre par un humain. Par conséquent, la traduction de ce format en une représentation graphique s'impose. Afin de visualiser graphiquement ce GCFD, nous incorporons l'outil *VCG*

dans le compilateur *LCC* modifié.

3.3.1- Vue d'ensemble de l'outil *VCG*

VCG est un outil de visualisation graphique d'un graphe [Sand95]. Il existe dans le domaine public. Le code source de la version V1.30 de *VCG* se trouve dans la référence [VCG1.3]. Cet outil opère sur une spécification d'un graphe écrite en langage "Graph Description Langage" (GDL) [Sand95]. Une fois que la spécification est compilée par *VCG*, ce dernier affiche le graphe correspondant. La Figure 3.7 décrit l'environnement de *VCG*.

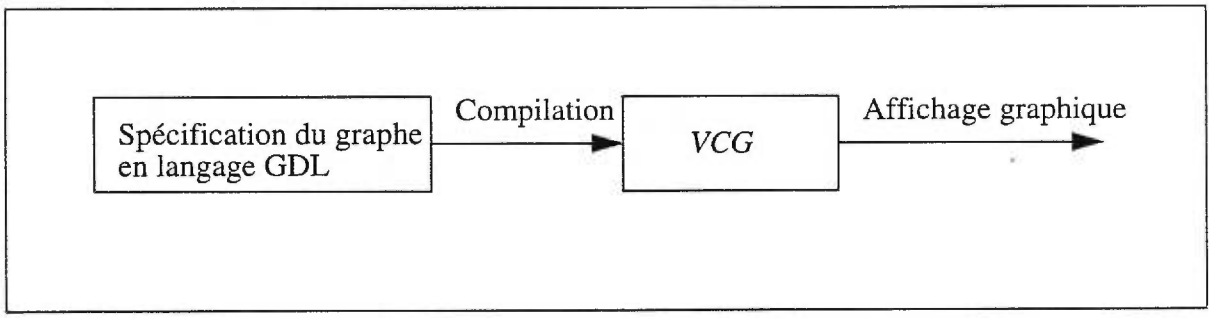


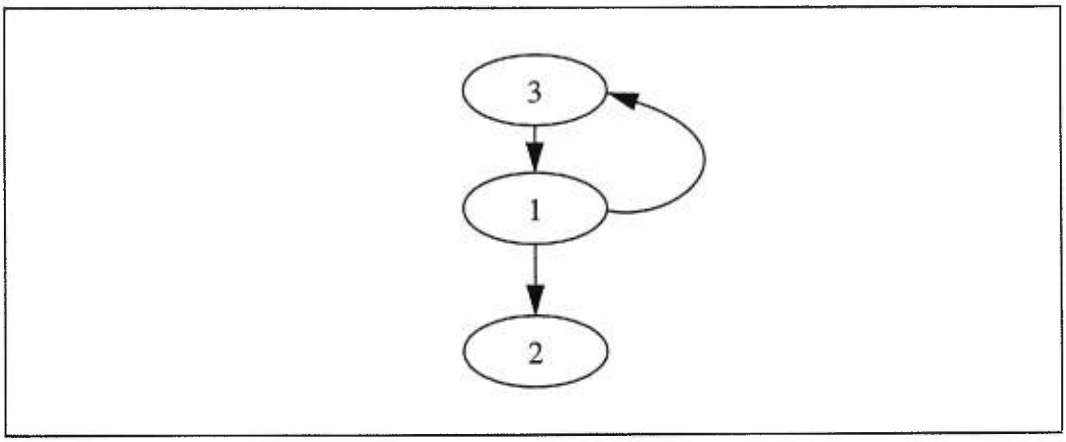
Figure 3.7 Environnement de l'outil VCG.

Le code suivant est un exemple d'une spécification d'un graphe en langage GDL:

```
graph: {

/* Définition des noeuds. */
     node: {title: "A" label: "1" shape: ellipse color:white}
     node: {title: "B" label: "2" shape: ellipse color:white}
     node: {title: "C" label: "3" shape: ellipse color:white}
/* Définition des arcs. */
     edge: {sourcename: "A" targetname: "B"}
     edge: {sourcename: "A" targetname: "C"}
     edge: {sourcename: "C" targetname: "A"}
}
```

L'affichage graphique correspondant à la spécification du graphe ci-dessus est le suivant:



3.3.2- Intégration de l'outil *VCG* dans le compilateur *LCC*

La Figure 3.8 décrit l'environnement du compilateur *LCC* modifié après intégration de l'outil *VCG*. Nous avons réalisé cette intégration au niveau *génération automatique* (voir Figure 3.8). À ce niveau, le compilateur *LCC* modifié génère un GCFD et la spécification de celui-ci en langage "Graph Description Language" (GDL). Ensuite, cette spécification est consommée par l'outil *VCG* pour visualiser graphiquement ce GCFD. Dans la version actuelle du compilateur *LCC* modifié, l'affichage textuel du GCFD est conservé.

Nous avons réalisé la génération automatique de la spécification en langage GDL du GCFD généré par le compilateur *LCC* modifié à l'aide de la fonction *vcg_gcfd*. L'appel à cette fonction se fait comme il est montré à la Figure 3.6.

La Figure 3.9 présente un exemple d'un graphe de contrôle et de flot de données produit par la version actuelle du compilateur *LCC* modifié après intégration de l'outil *VCG* dans celui-ci.

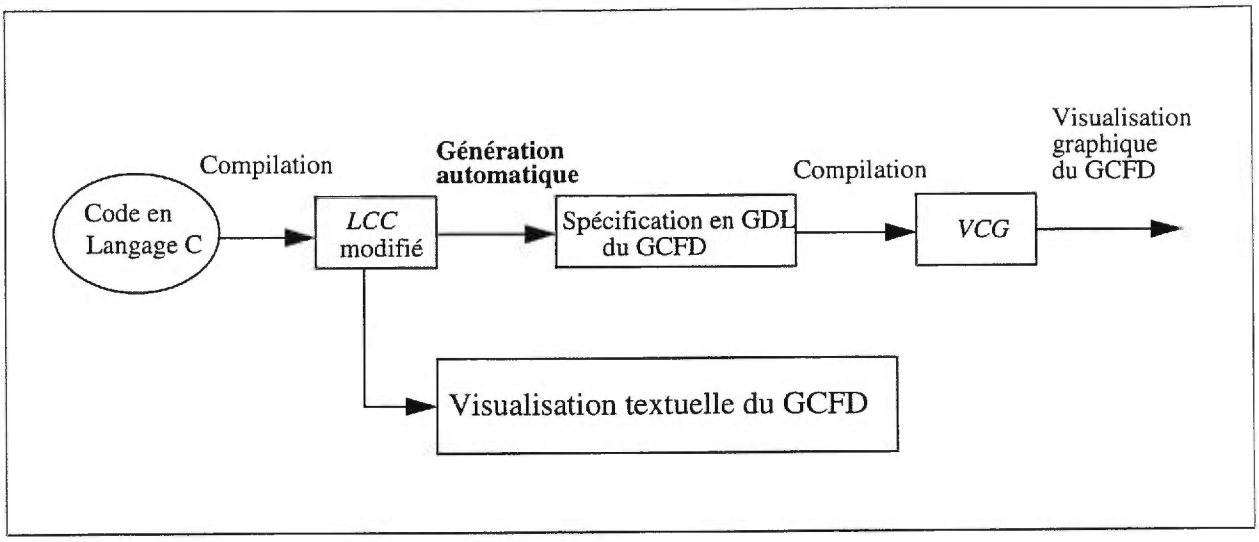
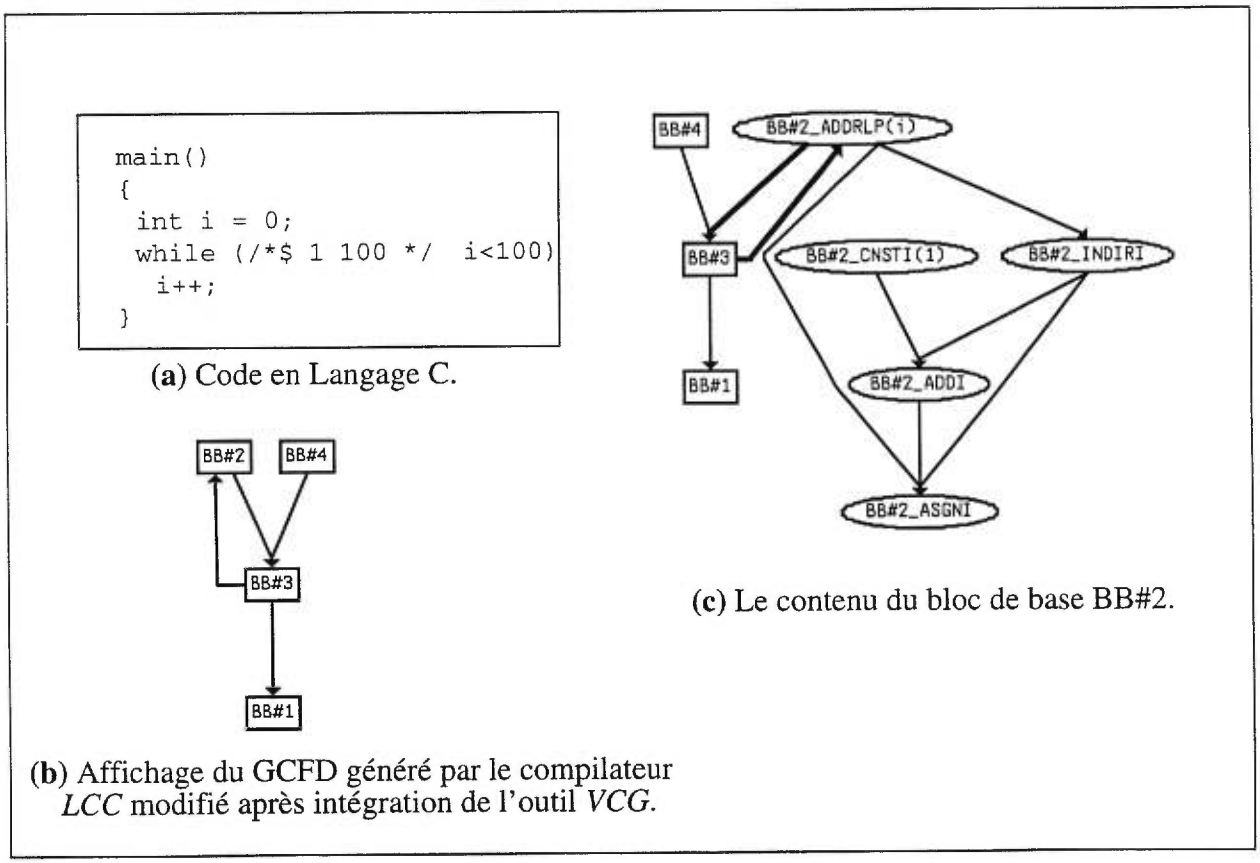


Figure 3.8 Intégration de l'outil VCG dans le compilateur LCC modifié.



(**a**) Code en Langage C.

(**b**) Affichage du GCFD généré par le compilateur *LCC* modifié après intégration de l'outil *VCG*.

(**c**) Le contenu du bloc de base BB#2.

Figure 3.9 Exemple de GCFD généré par le compilateur LCC modifié.

3.4- Conclusion

Dans ce chapitre, nous nous sommes concentrés sur la modification de l'interface *symbolic* du compilateur *LCC*. Nous avons résolu le problème de dépendances dans les blocs de base du graphe de contrôle et de flot de données (GCFD) généré par l'interface *symbolic* du compilateur *LCC*. Nous avons résolu le problème de la détection des boucles dans ce GCFD ainsi que la détermination des nombres minimal et maximal d'itérations de celles-ci. Afin de visualiser graphiquement le graphe de contrôle et de flot de données généré par l'interface *symbolic* du compilateur *LCC* modifié, nous avons développé une interface permettant de communiquer ce compilateur avec l'outil *VCG*.

Chapitre 4

Parallélisation des boucles imbriquées

Dans ce mémoire, tout programme à exécuter sur un système parallèle de type SIMD est décrit en langage séquentiel (plus précisément en Langage C). En plus, ce programme correspond à un traitement itératif (i.e., présence de boucles). Comme nous allons voir au Chapitre 5, nous avons besoin de partitionner ce programme sur les processeurs de ce système. Afin d'y arriver le besoin d'un compilateur parallèle s'impose. Dans notre cas, nous ne disposons pas de compilateur parallèle. En conséquence, ce partitionnement se fait par le programmeur. Toujours dans le cas où nous ne disposons pas de compilateur parallèle, le programmeur doit partitionner ce programme en des *sous-programmes* indépendants, c'est-à-dire en des *sous-programmes* dont l'exécution ne requiert pas de communications interprocesseurs.

Au long de ce travail, le mot *sous-programme* désigne le programme à exécuter sur un processeur.

En général, le partitionnement d'un programme en des *sous-programmes* indépendants peut ne pas être immédiat. Ceci découle du fait que le parallélisme dans ce programme peut être caché. Afin de faire apparaître ce parallélisme, des transformations de celui-ci sont requises comme nous allons voir dans ce chapitre.

Ce chapitre est une synthèse de la littérature sur la parallélisation d'un ensemble de boucles imbriquées. Dans la suite, nous appelons un ensemble de boucles imbriquées *nid de boucles* comme il est appelé dans [Dart93]. Notre objectif est d'étudier comment, à partir d'un nid de boucles, nous pouvons reconstruire un autre nid de boucles pour lequel le nombre de boucles pouvant être exécutées en parallèle est maximisé. Afin d'atteindre cet objectif, nous présentons un processus de parallélisation d'un nid de boucles qui se compose de trois étapes. La première étape est l'extraction des vecteurs des dépendances inter-itérations. La deuxième étape est la mise du nid

de boucles sous un format dit canonique, que nous allons définir dans la suite. Une fois ce format obtenu, la troisième étape consiste à appliquer une transformation appelée "Wavefront" sur celui-ci afin de construire le nid de boucles final.

Ce chapitre est organisé de la façon suivante. Dans la Section 1, nous donnons quelques définissons. La Section 2 examine les transformations d'un nid de boucles. Nous présentons la parallélisation d'un nid de boucles dans la Section 3.

4.1- Définitions

4.1.1- Nid de boucles

Un nid de boucles est un ensemble de boucles imbriquées. Si n est le nombre de boucles d'un nid, ce dernier est dit de profondeur n. Sauf indication contraire, nous désignons dans le reste de ce chapitre par nid de boucles la structure de programme suivante:

for $i_1 = l_1$ to u_1 do
  for $i_2 = l_2(i_1)$ to $u_2(i_1)$ do
    ⋮
    for $i_n = l_n(i_1, i_2, ..., i_{n-1})$ to $u_n(i_1, i_2, ..., i_{n-1})$ do
      {Instruction de calcul};

où $l_{i=2, ..., n}$ et $u_{i=2, ..., n}$ sont des fonctions affines, et l_1 et u_1 sont deux constantes.

4.1.2- Domaines d'itérations

Le domaine d'itérations d'un nid de boucles de profondeur n est un polyèdre convexe et fini dans l'espace d'itérations Z^n [Dart93], où Z est l'ensemble des entiers relatifs. Chaque noeud du polyèdre représente une itération de ce nid et il se caractérise par un vecteur d'indices $P = (p_1, ..., p_n)$, où p_i représente la valeur de l'indice correspondant à la i^{eme} boucle. $(p_1, ..., p_n)$ désigne, au long de ce chapitre, un vecteur colonne. Les boucles sont numérotées d'une façon croissante à partir de la boucle la plus externe. La Figure 4.1 présente un exemple de nid de boucles de profondeur deux, son domaine d'itérations et l'exécution séquentielle de celui-ci. Pour une exécution séquentielle, les itérations du nid de boucles s'exécutent dans un ordre lexicographique. Si $P = (p_1, ..., p_n)$ et $\overline{P} = (\overline{p}_1, ..., \overline{p}_n)$ sont deux itérations de celui-ci, alors

P sera exécutée avant $\bar{P}$ si et seulement si $\bar{P}$ est lexicographiquement supérieure à P, que l'on note aussi $\bar{P} >> P$. Donc, $\bar{P} >> P$ si et seulement si, $\bar{p}_1 > p_1$ ou, $\bar{p}_1 = p_1$ et $(\bar{p}_2, \ldots, \bar{p}_n) >> (p_2, \ldots, p_n)$.

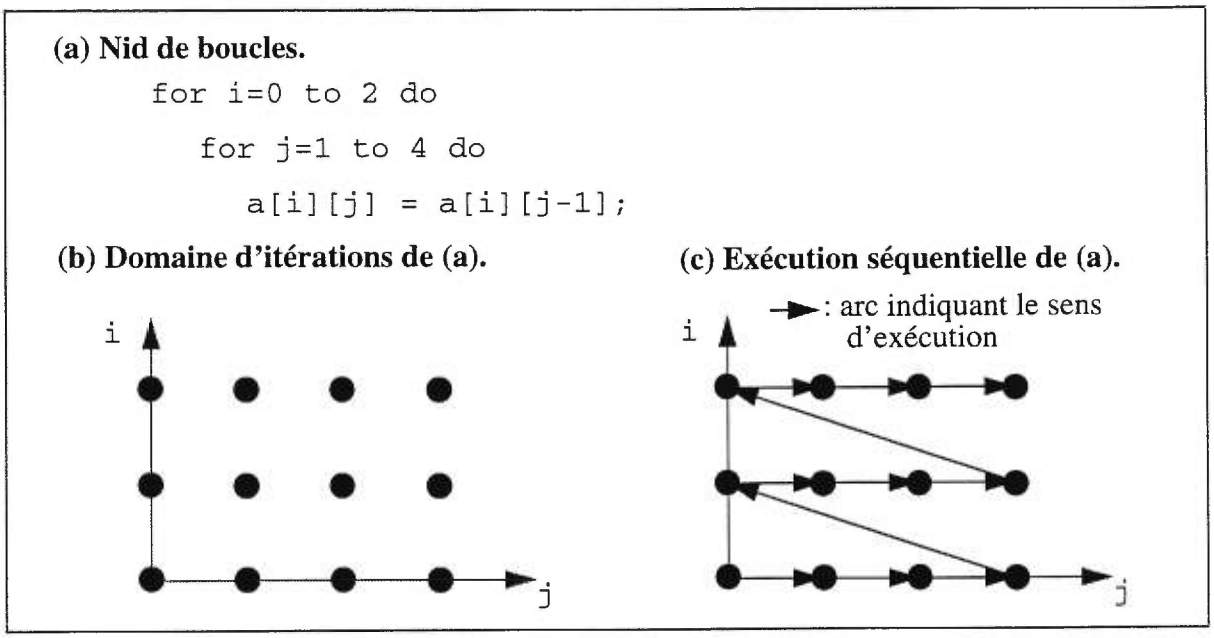


Figure 4.1 Exemple de nid de boucles, son domaine d'itérations, et son exécution séquentielle.

4.1.3- Vecteurs de dépendance

Soient $P = (p_1, \ldots, p_n)$ et $\bar{P} = (\bar{p}_1, \ldots, \bar{p}_n)$ deux itérations d'un nid de boucles de profondeur n telles que $\bar{P} >> P$. Au long de ce chapitre, nous désignons par *une itération P accède une adresse mémoire adr* le fait que l'exécution de cette itération requiert l'accès à *adr*. Si $\bar{P}$ et P accèdent à une même adresse mémoire *Adr*, si aucune autre itération entre P et $\bar{P}$ ne requiert un accès à *Adr*, et si au moins une de ces deux itérations est une écriture dans *Adr*, alors une dépendance existe entre $\bar{P}$ et P. Afin de conserver la sémantique du programme qui est donnée par son exécution séquentielle, alors avant de transformer le nid de boucles original, un arc de dépendance doit être ajouté de P vers $\bar{P}$. L'ajout de cet arc permet de bloquer l'exécution de $\bar{P}$ jusqu'à ce que P soit exécutée.

Si un arc de P vers $\bar{P}$ existe, alors $D = \bar{P} - P$ est appelé vecteur de dépendance, et $D = (d_1, \ldots, d_n) = (\bar{p}_1 - p_1, \ldots, \bar{p}_n - p_n)$. Si tous les d_i sont constants, D est dit vecteur de distances. L'ensemble des vecteurs des dépendances, dénoté S, est défini comme suit:

$$S = \{D | (D = \overline{P} - P) \text{ et un arc de } P \text{ à } \overline{P} \text{ existe}\}.$$

Par exemple, pour le nid de boucles de la Figure 4.2, l'écriture dans l'adresse de l'élément $a[i][j]$ réfère en lecture à l'adresse de l'élément $a[i][j-1]$. Dans le cas de l'exécution séquentielle, l'écriture dans l'adresse de $a[i][j-1]$ se fait avant celle dans l'adresse de $a[i][j]$. Ainsi, l'écriture dans l'adresse de $a[i][j]$ doit utiliser le résultat de l'écriture dans l'adresse de $a[i][j-1]$. Cela justifie l'existence du vecteur de dépendance $D = (0, 1)$.

(a) Nid de boucles.

```
for i=0 to 2 do
  for j=1 to 4 do
    a[i][j] = a[i][j-1];
```

(b) Dépendances inter-itérations pour (a).

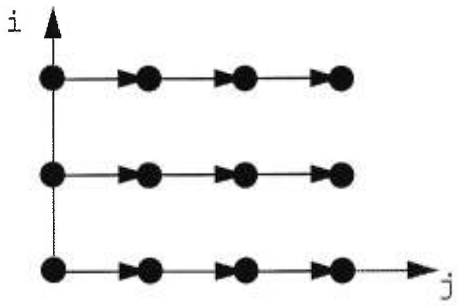


Figure 4.2 Exemple de nid de boucles et les dépendances inter-itérations pour celui-ci.

Quand un vecteur de dépendance D n'est pas constant, alors celui-ci représente une famille de vecteurs de distances notée $\mathcal{F}(D)$. Cette famille est définie comme suit:

$$\mathcal{F}(D) = \{E = (e_1, \ldots, e_n) \,|\, (e_i \in Z) \text{ et } (b_i \leq e_i \leq B_i)\}, \text{ où}$$

$$(b_i \in Z \cup \{-\infty\}) \text{ et } (B_i \in Z \cup \{+\infty\}).$$

Notons que si D est constant alors $\mathcal{F}(D) = D$. Des discussions sur les techniques d'extraction des vecteurs des dépendances se trouvent dans [IriT88], [TseW90] et [MayH91].

4.1.4- Positivité lexicographique d'un vecteur de dépendance

Pour une exécution séquentielle d'un nid de boucles, les itérations s'exécutent dans un ordre lexicographique. Comme les dépendances constituent une contrainte supplémentaire pour une exécution séquentielle d'un nid de boucles, alors tous les vecteurs de dépendance sont lexicographiquement positifs. La positivité lexicographique d'un vecteur de dépendance est définie comme suit.

Définition 4.1 *Soit D un vecteur de dépendance. D est lexicographiquement positif, noté $D>>0$, si et seulement si $\forall E \in \mathcal{F}(D)$: $E>>0$. D est lexicographiquement non-négatif si et seulement si $\forall E \in \mathcal{F}(D)$: $E \geq 0$.*

Théorème 4.1 *Soit D un vecteur de dépendance. $D>>0$ si et seulement si $b_1 > 0$ ou, ($b_1 = 0$ et $(e_2, \ldots, e_n) >> 0$).*

La preuve de ce théorème se fait par une induction mathématique dans un sens, et par l'utilisation de la Définition 4.1 dans l'autre sens. Une preuve complète se trouve dans [Wolf92]. ❑

Théorème 4.2 *Soit D un vecteur de dépendance. $D>>0$ si et seulement si $\exists i$: $((d_i > 0)$ et $\forall j < i$: $(d_j \geq 0))$.*

La preuve de ce théorème découle du Théorème 4.1. ❑

4.2- Transformations d'un nid de boucles

Les transformations d'un nid de boucles consistent à réarranger l'ordre d'exécution des itérations du nid de boucles original afin de faire apparaître du parallélisme caché et, par la suite, à réécrire un nouveau nid de boucles pour lequel le nombre de boucles s'exécutant en parallèle est maximisé. Ces transformations se divisent en deux groupes: transformations unimodulaires, et transformations non-unimodulaires. Dans le reste de ce chapitre, nous ne mettons l'accent que sur les transformations unimodulaires.

Cette section se compose de trois sous-sections. Dans la première sous-section, nous présentons le groupe des transformations unimodulaires, ainsi que quelques-unes des transformations élémentaires les plus fréquemment utilisées dans la littérature. À la deuxième sous-section, nous discutons de la validité d'une transformation. La troisième sous-section présente les étapes de la génération du code d'un nid de boucles après sa transformation.

4.2.1- Transformations unimodulaires

Elles sont un cas particulier de transformations de boucles. Elles se caractérisent par le fait qu'elles peuvent être représentées par des matrices unimodulaires [Bane88] et [Wolf92]. Une matrice unimodulaire T a trois propriétés. Premièrement, T est une matrice carrée. Elle permet de transformer un nid de boucles de profondeur n en un autre de profondeur n. Deuxièment, T a tous ses éléments entiers. Ainsi, elle transforme chaque vecteur d'entiers en un autre à éléments entiers. Troisièment, la valeur absolue du déterminant de T vaut un. Grâce à ces trois propriétés, le produit de deux matrices unimodulaires, et l'inverse d'une matrice unimodulaire, sont des matrices unimodulaires. Par conséquent, la combinaison de deux transformations unimodulaires, et l'inverse d'une transformation unimodulaire, sont aussi des transformations unimodulaires.

Ces transformations ont attiré l'attention des chercheurs dans le domaine de la parallélisation du code par le fait qu'elles transforment un domaine d'itérations en un autre qui ne contient pas de trous. Les transformations non-unimodulaires produisent un domaine d'itérations qui ne sera parcouru que partiellement; c'est-à-dire qu'il existe dans ce domaine des itérations pour lesquelles les instructions de calcul ne seront pas exécutées. Par exemple, pour le nid de boucles de la Figure 4.3 (c) que nous obtenons en appliquant l'algorithme présenté à la Section 4.2.3, l'itération $(4, 1)$ n'exécute pas une instruction de calcul puisque $a[5/2][3/2]$ n'est pas un élément du tableau a. Un autre avantage des transformations unimodulaires est la facilité de déterminer les bornes inférieures et supérieures sur les valeurs des compteurs des itérations des boucles du nid transformé [Wolf92].

Les transformations unimodulaires les plus connues dans la littérature sont la Permutation, l'Inversion, la "Skewing", et la "Wavefront". Les matrices représentantes pour les trois premières transformations peuvent être vues comme le résultat de la modification de certaines lignes de la

matrice identité I. Dans le reste de cette section, nous présentons les transformations Permutation, Inversion et "Skewing", ainsi que leur matrices représentantes. La transformation "Wavefront" est présentée à la Section 4.3.

4.2.1.1- Permutation

La Permutation, comme son nom l'indique, consiste à permuter deux ou plusieurs boucles. Elle est représentée par une matrice T de taille $n \times n$, où n est la profondeur du nid de boucles. Cette matrice est le résultat de la permutation de certaines lignes de la matrice identité I. Par exemple, la matrice T qui représente la permutation des première et deuxième boucles d'un nid de boucles de profondeur deux est $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. La Figure 4.4 présente un exemple d'application de la transformation Permutation. Le nid de boucles obtenu après cette transformation est donné à la Figure 4.4 (b). Pour obtenir le code de celui-ci, nous appliquons l'algorithme présenté à la Section 4.2.3. L'application des étapes de cet algorithme est comme suit:

Étape 1: Les bornes inférieures et supérieures sur les valeurs des compteurs I_1 et I_2 sont:

$$0 \le I_1 \le 7 \text{ et } 1 \le I_2 \le 4 .$$

Étape 2: Par

$$\begin{bmatrix} J_1 \\ J_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} I_1 \\ I_2 \end{bmatrix},$$

nous obtenons:

$$J_1 = I_2 \text{ et } J_2 = I_1 .$$

Étape 3: À partir de l'étape 1 et de l'étape 2, nous obtenons les bornes inférieures et supérieures sur les valeurs des compteurs J_1 et J_2 comme suit:

$$1 \le J_1 \le 4 \text{ et } 0 \le J_2 \le 7 .$$

Étape 4: En remplaçant le compteur d'itérations de chaque boucle **for**, du nid de boucles original, par son compteur dans le nid de boucles après transformation, et en ajoutant les bornes inférieures et supérieures sur la valeur de ce compteur, nous obtenons le code suivant:

```
for (J1=1; J1<=4; J1++)
  for (J2=0; J2<=7; J2++)
```

Étape 5: Par

$$\begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} J_1 \\ J_2 \end{bmatrix},$$

nous obtenons:

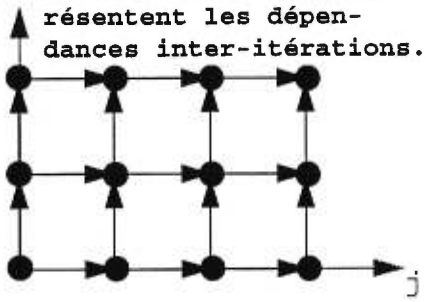
$$I_1 = J_2 \text{ et } I_2 = J_1.$$

Étape 6: Le corps du nid de boucles original est `a[I1][I2] = a[I1][I2-1]`. En remplaçant I_1 et I_2 par leurs expressions obtenues à l'étape 5, nous obtenons le corps, du nid de boucles final, suivant: `a[J2][J1] = a[J2][J1-1]`.

(a)Nid de boucles original.

```
for i=1 to 3 do
  for j=1 to 4 do
    a[i][j] = a[i][j-1] + a[i-1][j];
```

Domaine d'itérations de (a). Les arcs représentent les dépendances inter-itérations.



(b) Nid de boucles après une transformation unimodulaire, T_u, appliquée à (a).

```
for r=2 to 7 do
  for s=max (1, r-4) to min(3, r-1) do
    a[s][r-s] = a[s][r-s-1] + a[s-1][r-s];
```

$$T_u = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

(c) Nid de boucles après une transformation non-unimodulaire, T_{nu}, appliquée à (a).

$$T_{nu} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

```
for r=2 to 7 do
  for s=max (-3, r-8) to min(2, r-2) do
    a[(r+s)/2][(r-s)/2]=a[(r+s)/2][(r-s-2)/2]+a[(r+s-2)/2][(r-s)/2];
```

Figure 4.3 Transformations unimodulaires vs. non-unimodulaires.

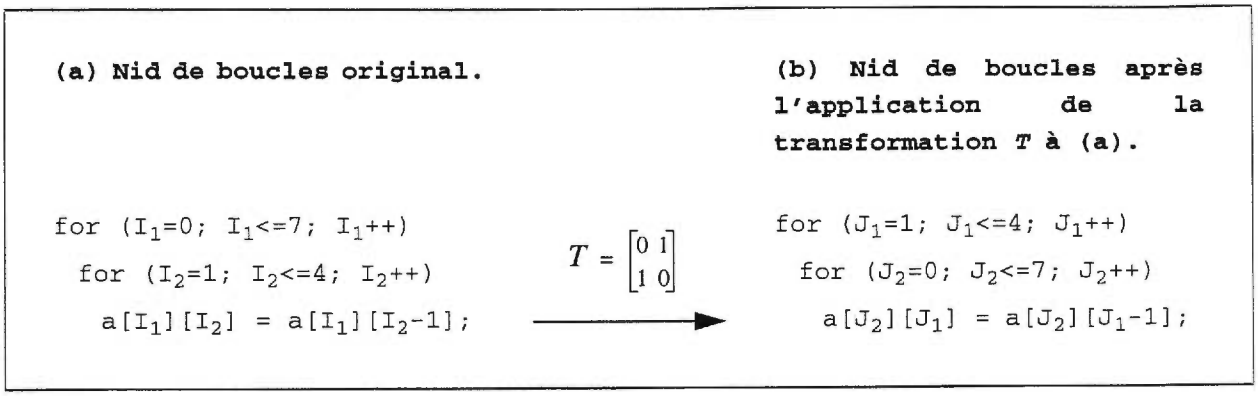


Figure 4.4 Exemple d'application d'une transformation Permutation à un nid de boucles.

4.2.1.2- Inversion

Ce type de transformation consiste à inverser une ou plusieurs boucles d'un nid de boucles de profondeur n. L'inversion d'une boucle correspond à inverser le signe de son compteur d'itérations. Cette transformation peut être représentée par une matrice T de taille $n \times n$. Cette matrice est le résultat de la multiplication de certaines lignes de la matrice identité I par -1. Par exemple, l'inversion de la deuxième boucle d'un nid de boucles de profondeur deux est représentée par la matrice $T = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. La Figure 4.5 donne un exemple d'application d'une transformation Inversion. La Figure 4.5 (b) présente le nid de boucles obtenu après cette transformation. Nous obtenons le code de celui-ci en appliquant l'algorithme présenté à la Section 4.2.3.

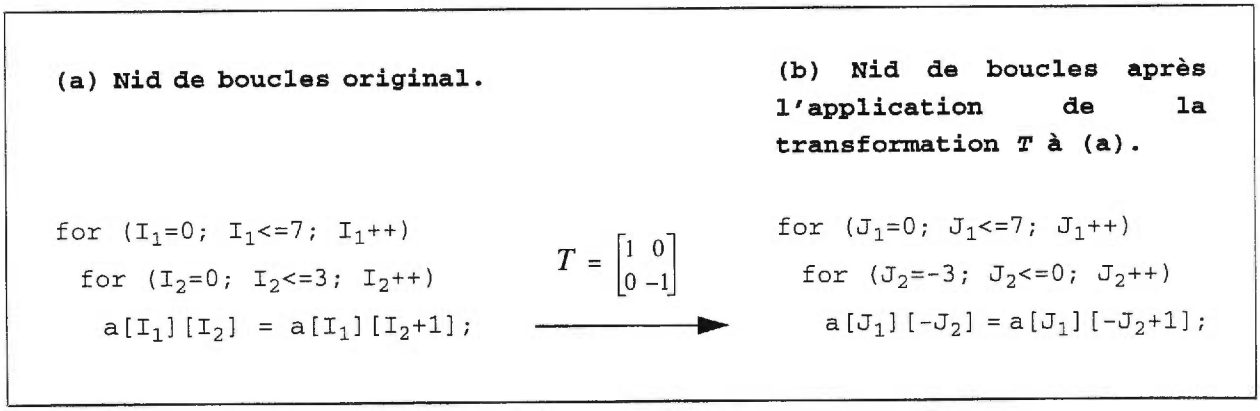


Figure 4.5 Exemple d'application d'une transformation Inversion à un nid de boucles.

4.2.1.3- "Skewing"

Elle consiste à incliner par un facteur f la j^{eme} boucle par rapport à celle de numéro i, avec $i < j$. L'application de cette transformation à un nid de boucles de profondeur n transforme l'itération $P_1 = (p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n)$ à l'itération $P_2 = (p_1, \ldots, p_i, \ldots, p_j + f \cdot p_i, \ldots, p_n)$. Cette transformation peut être représentée par une matrice T de taille $n \times n$. Cette matrice est la matrice identité I où la valeur de l'élément $I_{j,i}$ est remplacé par f. Par exemple, l'inclinaison par un facteur f de la deuxième boucle d'un nid de boucles de profondeur deux par rapport à la première boucle est représentée par la matrice $T = \begin{bmatrix} 1 & 0 \\ f & 1 \end{bmatrix}$. Des discussions sur le choix du facteur f se trouvent dans [WolL91] et [Wolf92]. La Figure 4.6 illustre un exemple d'application d'une transformation "Skewing" dont le facteur f vaut 1. La Figure 4.6 (b) présente le nid de boucles obtenu après cette transformation. Nous obtenons le code de celui-ci en appliquant l'algorithme donné à la Section 4.2.3.

(a) Nid de boucles original.

```
for (I1=1; I1<=8; I1++)
  for (I2=0; I2<=3; I2++)
    a[I1][I2] = a[I1-1][I2+1];
```

$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ ⟶

(b) Nid de boucles après l'application de la transformation T à (a).

```
for (J1=1; J1<=8; J1++)
  for (J2=J1; J2<=J1+3; J2++)
    a[J1][J2-J1] = a[J1-1][J2-J1+1];
```

Figure 4.6 Exemple d'application d'une transformation "Skewing" à un nid de boucles.

4.2.2- Validité d'une transformation

L'application d'une transformation à un nid de boucles ne doit pas changer sa sémantique. Celle-ci correspond au résultat de son exécution séquentielle. Une transformation est dite valide si une lecture à partir d'une adresse mémoire et une écriture dans cette adresse doivent être faites dans le même ordre que dans le nid de boucles original. Aussi, si deux écritures requièrent une même adresse mémoire, alors l'ordre d'exécution de celles-ci après application de la transformation doit être le même que celui dans le cas de l'exécution séquentielle du nid original.

Plus formellement, la légalité d'une transformation peut être définie comme suit. Soit T la matrice qui représente la transformation à appliquer à un nid de boucles, et soient P et $\bar{P}$ deux itérations du domaine d'itérations de ce nid. Si P et $\bar{P}$ accèdent une même adresse mémoire, qu'au moins une de ces deux itérations est une écriture, et que P s'exécute avant $\bar{P}$ dans le cas de l'exécution séquentielle, alors $T \cdot P$ doit être exécutée avant $T \cdot \bar{P}$ dans le domaine d'itération du nid après transformation. En conséquence, si $D = \bar{P} - P$ est le vecteur de dépendance entre P et $\bar{P}$, alors $T \cdot D$ est le vecteur de dépendance entre $T \cdot P$ et $T \cdot \bar{P}$. En effet, comme $T \cdot P$ s'exécute avant $T \cdot \bar{P}$, alors le vecteur de dépendance entre $T \cdot P$ et $T \cdot \bar{P}$ est $(T \cdot \bar{P} - T \cdot P)$. Puisque T est une transformation linéaire, alors $T \cdot \bar{P} - T \cdot P = T \cdot (\bar{P} - P) = T \cdot D$. D'où, si T est valide, alors $T \cdot D$ est lexicographiquement positif puisque $T \cdot \bar{P}$ est lexicographiquement supérieure à $T \cdot P$. Basé sur cette formulation, nous énonçons le théorème de la validité d'une transformation ([Dowl90] et [Wolf92]) comme suit.

Théorème 4.3 *Soit T la matrice qui représente une transformation d'un nid de boucles. Cette transformation est valide si et seulement si* $\forall D \in S : (T \cdot D) >> 0$, *où S est l'ensemble des vecteurs des dépendances, et la relation ">>" signifie lexicographiquement supérieure à.* ❑

4.2.3- Génération du code du nid de boucles après une transformation

Soient $(I_1, \ldots, I_n)$ le vecteur des compteurs des boucles du nid original, T la matrice qui représente la transformation à apporter à celui-ci, et $(J_1, \ldots, J_n)$ le vecteur des compteurs des boucles pour le nid de boucles transformé. La génération du code du nid de boucles après transformation peut être faite selon l'algorithme décrit à la Figure 4.7.

Algorithme:

Entrées: Un nid de boucles dont le vecteur des compteurs est $(I_1, \ldots, I_n)$, et une transformation T à appliquer à celui-ci.

Sortie: Un nid de boucles dont le vecteur des compteurs est $(J_1, \ldots, J_n)$.

Début:

1: Établir les bornes inférieures et supérieures sur la valeur de I_k, où $k \in [1, n]$.

2: Exprimer chaque J_k en fonction des I_l. $(J_1, \ldots, J_n) = T \cdot (I_1, \ldots, I_n)$.

3: Calculer les bornes inférieures et supérieures sur la valeur de chaque J_k.

4: Pour chaque boucle **for**, remplacer son compteur d'itérations (i.e., I_k) par le nouveau compteur (i.e., J_k). Les bornes inférieures et supérieures sur la valeur de celui-ci sont déterminées à l'étape 3.

5: Retrouver les I_k en fonction des J_l. $(I_1, \ldots, I_n) = T^{-1} \cdot (J_1, \ldots, J_n)$.

6: Le corps du nouveau nid de boucles s'obtient à partir de celui du nid original en remplaçant chaque I_k par son expression en fonction des J_l. Cette expression est déterminée à l'étape 5.

Fin.

Figure 4.7 Génération du code résultat d'une transformation appliquée à un nid de boucles.

4.3- Parallélisation d'un nid de boucles

La parallélisation d'un nid de boucles consiste à maximiser le nombre de boucles parallèles, dit aussi degré de parallélisme, en lui appliquant des transformations. Une boucle est dite parallèle si toutes ses itérations peuvent être exécutées simultanément; ceci n'est possible que s'il n'y pas de dépendance entre ses itérations. Le processus de parallélisation d'un nid de boucles peut être décrit par les trois étapes suivantes.

La première étape consiste à extraire les vecteurs de dépendance pour le nid de boucles original. Ces vecteurs servent comme des données pour la deuxième étape.

La deuxième étape est la mise du nid de boucles sous un format canonique. La définition

du format canonique (voir aussi [Wolf92]) d'un nid de boucles est la suivante:

Définition 4.2 *Un nid de boucle est dit sous un format canonique si et seulement si toutes les permutations possibles à appliquer à celui-ci sont valides.*

L'objectif de la construction d'un nid de boucles sous un format canonique est de s'assurer que l'application de la transformation qui permet sa parallélisation est toujours valide. Une condition nécessaire et suffisante pour qu'un nid de boucles soit sous un format canonique est que les composantes de tous les vecteurs de dépendance soient supérieures ou égales à zéro [Wolf92].

La troisième étape est la construction d'un nid de boucles dont le degré de parallélisme est maximisé en appliquant la transformation "Wavefront" (voir sa définition dans la suite) sur le nid obtenu dans l'étape 2.

Dans le reste de cette section, nous continuons les discussions sur les deuxième et troisième étapes du processus de parallélisation d'un nid de boucles, mentionné ci-dessus, dans le cas où les vecteurs de dépendance sont constants et dans le cas où un des vecteurs de dépendances n'est pas constant.

4.3.1- Parallélisation dans le cas où les vecteurs de dépendances sont constants

Dans ce cas, la deuxième étape du processus de parallélisation d'un nid de boucles consiste à rendre le nid de boucles sous un format canonique. Plus précisément, elle rend les composantes de tous les vecteurs de dépendance non-négatives. À cette fin, Wolf et Lam [WolL91] proposent un algorithme qui permet de réaliser cette étape. Cet algorithme consiste en une combinaison des transformations élémentaires Permutation, Inversion, et "Skewing". Nous avons présenté ces transformations élémentaires à la Section 4.2.1.

Une fois que le nid de boucles, de profondeur n, est mis sous un format canonique, la troisième étape consiste à appliquer la transformation "Wavefront" à celui-ci afin de maximiser le degré de parallélisme. Cette transformation est représentée par la matrice T, de taille $n \times n$, suivante:

$$T = \begin{bmatrix} 1 & 1 & \dots & 1 & 1 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}.$$

Dans ce cas, Wolf et Lam [WolL91] ont démontré que l'application de cette transformation permet la génération d'un nid de boucles dont le degré de parallélisme est $n-1$.

Dans la suite, nous présentons un exemple d'application de la transformation "Wavefront". La Figure 4.3 (a) présente un nid de boucles de profondeur deux. Les vecteurs de dépendance de celui-ci sont: (0, 1) et (1, 0). Les composantes de ces deux vecteurs sont supérieures ou égales à zéro. En conséquence, ce nid de boucles est sous un format canonique. En examinant le domaine d'itérations de celui-ci (voir Figure 4.3), nous remarquons qu'aucune des deux boucles ne peut être exécutée en parallèle. En appliquant la transformation "Wavefront" $T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ à ce nid, nous obtenons celui présenté à la Figure 4.3 (b). Le code du nid de boucles de cette figure s'obtient par application de l'algorithme présenté à la Section 4.2.3. Dans ce code, la boucle la plus externe est séquentielle, mais celle la plus interne est parallèle. Le degré de parallélisme ainsi obtenu vaut un.

4.3.2- Parallélisation dans le cas où un des vecteurs de dépendance n'est pas constant

Dans ce cas, l'objectif de la deuxième étape du processus de parallélisation d'un nid de boucles est la construction d'un ensemble de nids de boucles tels que ces derniers soient sous un format canonique. Chacun de ces nids de boucles doit être le plus profond sous nid du nid de boucles original. Leur construction se fait en parcourant le nid original de la boucle la plus externe vers la boucle la plus interne. Pour cette fin, un algorithme est proposé par Wolf et Lam [WolL91].

Une fois l'ensemble des nids de boucles construit, la troisième étape du processus de parallélisation d'un nid de boucles consiste à appliquer la transformation "Wavefront" sur chacun de ceux-ci, de la même façon que ce qui a été présenté à la Section 4.3.1.

Chapitre 5

Estimation des performances de machines de type SIMD

Comme nous l'avons mentionné au Chapitre 2, deux types de méthodes se présentent pour estimer les performances d'un système: les méthodes basées sur la simulation et les méthodes statiques. Les méthodes basées sur la simulation ne s'appliquent pas tant qu'un modèle exécutable du système ne soit pas établi. Les méthodes statiques peuvent être utilisées lors des premières phases de la conception du système ainsi qu'après sa fabrication.

Dans ce chapitre, nous examinons une manière d'estimer les performances de machines de type SIMD. Cette dernière se base sur l'application de la méthode statique d'estimation des performances d'un système vue au Chapitre 2. Par la suite, nous présentons un outil informatique d'estimation des performances de processeur parallèle de type "Very Long Instruction Word" (VLIW) appelé *C_PerfEstim*. Ensuite, nous introduisons quelques applications fréquemment utilisées dans le domaine du traitement d'images. Finalement, nous appliquons l'outil *C_PerfEstim* à l'estimation de performance d'une machine de type SIMD appelée *PULSE V1*, destinée pour des applications réelles.

5.1- Application d'une méthode statique pour estimer les performances de machines SIMD

Dans cette section, nous nous intéressons à l'estimation de performance de machines de type SIMD. Pour cette fin, nous appliquons une méthode statique vue au Chapitre 2. La Figure 5.1 décrit l'application de cette méthode. À partir de la description de l'architecture de cette machine et du programme correspondant à la tâche à réaliser par celle-ci, nous estimons des bornes inférieure et supérieure sur le temps d'exécution.

Comme nous l'avons vu au Chapitre 2, les algorithmes d'estimation de performance d'un système ont besoin d'un graphe de contrôle et de flot de données (GCFD). Nous rappelons qu'un GCFD est une représentation intermédiaire d'un programme donné. Pour traduire automatiquement un programme en un GCFD, l'utilisation d'un compilateur s'impose.

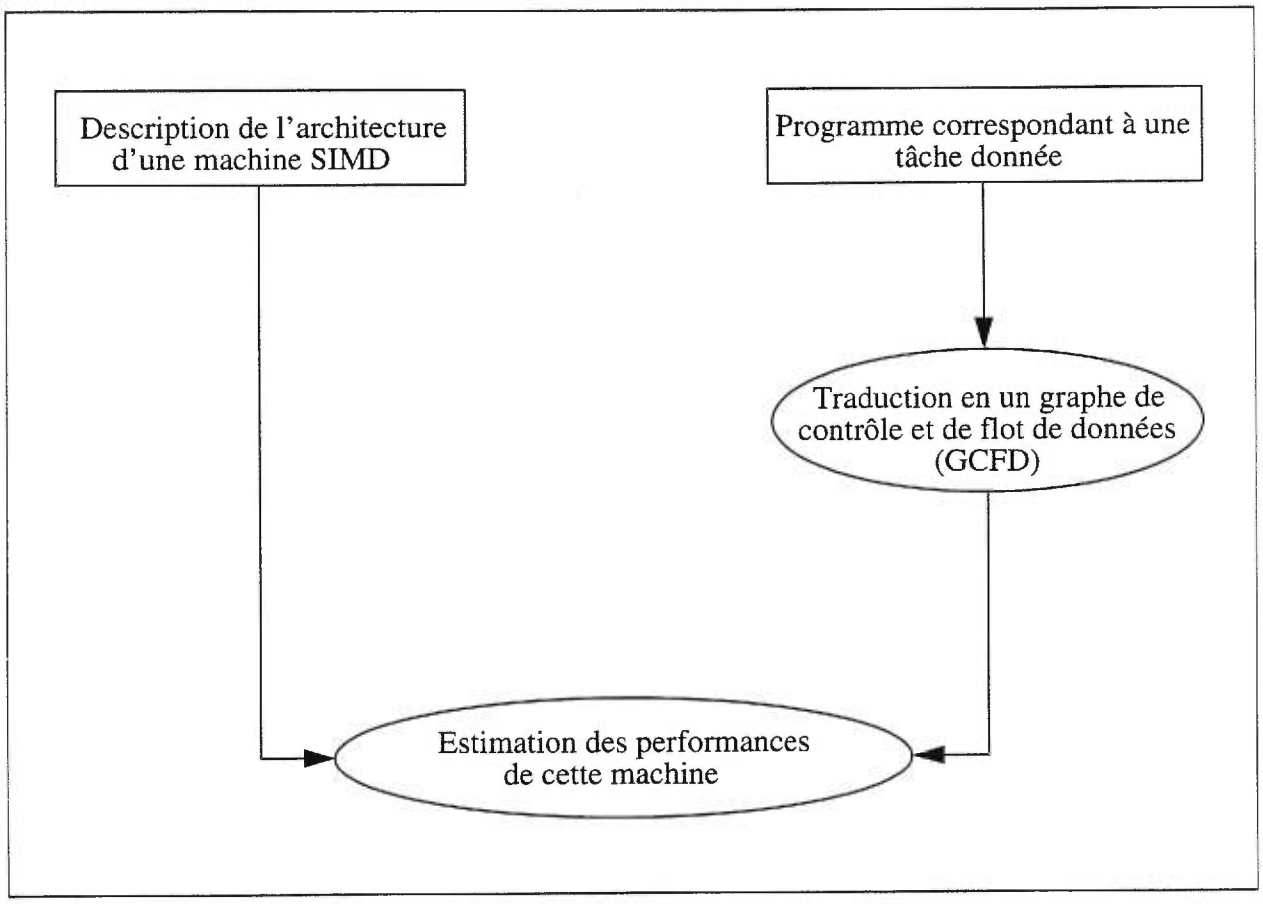


Figure 5.1 Description des étapes de l'application d'une méthode statique à l'estimation des performances d'une machine SIMD.

Pour traduire un programme à exécuter sur une machine SIMD en un graphe de contrôle et de flot de données (GCFD), nous avons besoin d'un compilateur parallèle. Dans notre cas, nous ne disposons pas de compilateur parallèle.

Comme nous l'avons mentionné au Chapitre 1, la programmation de machine de type SIMD se caractérise par l'écriture d'un seul programme s'exécutant par ses processeurs sous le

contrôle d'une seule unité de commande appelée contrôleur; tous les processeurs reçoivent la même instruction diffusée par le contrôleur, mais opèrent sur des données provenant de flux de données distincts. Par conséquent, si on suppose que les processeurs de cette machine sont identiques, nous pouvons considérer que l'estimation de performance de cette machine revient à estimer les performances d'un de ses processeurs. Notons que ces derniers peuvent être des processeurs parallèles de type VLIW ayant des ressources de type "pipeline".

Pour estimer les performances d'un des processeurs d'une machine SIMD, nous avons besoin de déterminer le programme que celui-ci va exécuter. Puisque nous ne disposons pas d'un compilateur parallèle, nous proposons le partitionnement du programme à exécuter sur cette machine en sous-programmes indépendants, c'est-à-dire en sous-programmes tels que leur exécution ne doit pas requérir des communications inter-processeurs. Toujours dans le contexte où nous ne disposons pas d'un compilateur pour machines de type SIMD, si nous n'arrivons pas à partitionner ce programme en un ensemble de sous-programmes indépendants, nous proposons de l'exécuter sur un des processeurs de cette machine. Notons que dans certains cas, le partitionnement d'un programme en sous-programmes qui ne sont pas indépendants peut dégrader les performances d'une machine SIMD.

En général, le partitionnement d'un programme en des sous-programmes indépendants peut ne pas être immédiat. Ceci découle du fait que le parallélisme dans le programme original peut être caché. En examinant par exemple le code suivant écrit en Langage C:

```
for (i = 1; i <= 3; i++)
   for (j = 1; j <= 4; j++)
      a[i][j] = a[i][j-1] + a[i-1][j];
```

Nous remarquons qu'aucune des deux boucles n'est parallélisable à cause des dépendances inter-itérations. Ces dernières sont induites par la référence aux éléments `a[i][j]` et `a[i][j-1]` pour la boucle la plus interne et par la référence aux éléments `a[i][j]` et `a[i-1][j]` pour la boucle externe. Mais, si on applique des transformations à ce code telles que vues au Chapitre 4, nous pouvons reconstruire un autre code formé de deux boucles imbriquées pour lequel la boucle la plus interne est parallélisable. En effet, en appliquant à ce code la transformation "Wavefront" représentée par la matrice T suivante (pour plus de détails voir le Chapitre 4):

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix},$$

nous obtenons le code, sémantiquement équivalent au précédent, suivant:

```
for (r = 2; r <= 7; r++)
   for (s = max (1, r-4); s <= min (3, r-1); s++)
      a[s][r-s] = a[s][r-s-1] + a[s-1][r-s];
```

Dans ce code, la boucle la plus interne est parallélisable. Par conséquent, chaque processeur d'une machine parallèle de type SIMD peut exécuter un code de la forme:

```
for (r = 2; r <= 7; r++)
   Une instruction de calcul d'un élément du tableau a;
```

Maintenant que nous avons examiné la détermination du programme à exécuter sur un processeur d'une machine SIMD, nous pouvons appliquer la méthode statique vue au Chapitre 2 pour estimer les performances de celui-ci. Pour cela, nous avons développé un outil d'estimation de performance d'un processeur. Cet outil est appelé *C_PerfEstim*. Nous le présentons dans la section suivante.

5.2- Présentation de l'outil *C_PerfEstim*

C_PerfEstim ("C Langage Performance Estimation") est un outil d'estimation de performance d'un processeur. Ce dernier peut avoir une architecture séquentielle ou parallèle de type "Very Long Instruction Word" (VLIW) avec des ressources de type "pipeline". Cet outil est une implémentation de la méthode statique d'estimation de performance d'un système vue au Chapitre 2. L'outil est implémenté en Langage C. La Figure 5.2 donne une vue d'ensemble de l'outil *C_PerfEstim*. Les entrées de l'outil sont la description de l'architecture d'un processeur et un graphe de contrôle et de flot de données (GCFD). Ce dernier correspond au programme à exécuter sur ce processeur. L'implémentation de ce programme est décrite en Langage C. Les sorties de l'outil *C_PerfEstim* sont une description de l'ordonnancement et d'allocation des ressources, ainsi que deux bornes, inférieure et supérieure, sur le temps d'exécution de ce programme sur ce processeur.

Nous examinons dans le reste de cette section la description de l'architecture d'un processeur dans l'outil *C_PerfEstim* et le graphe de contrôle et de flot de données qui alimente cet outil.

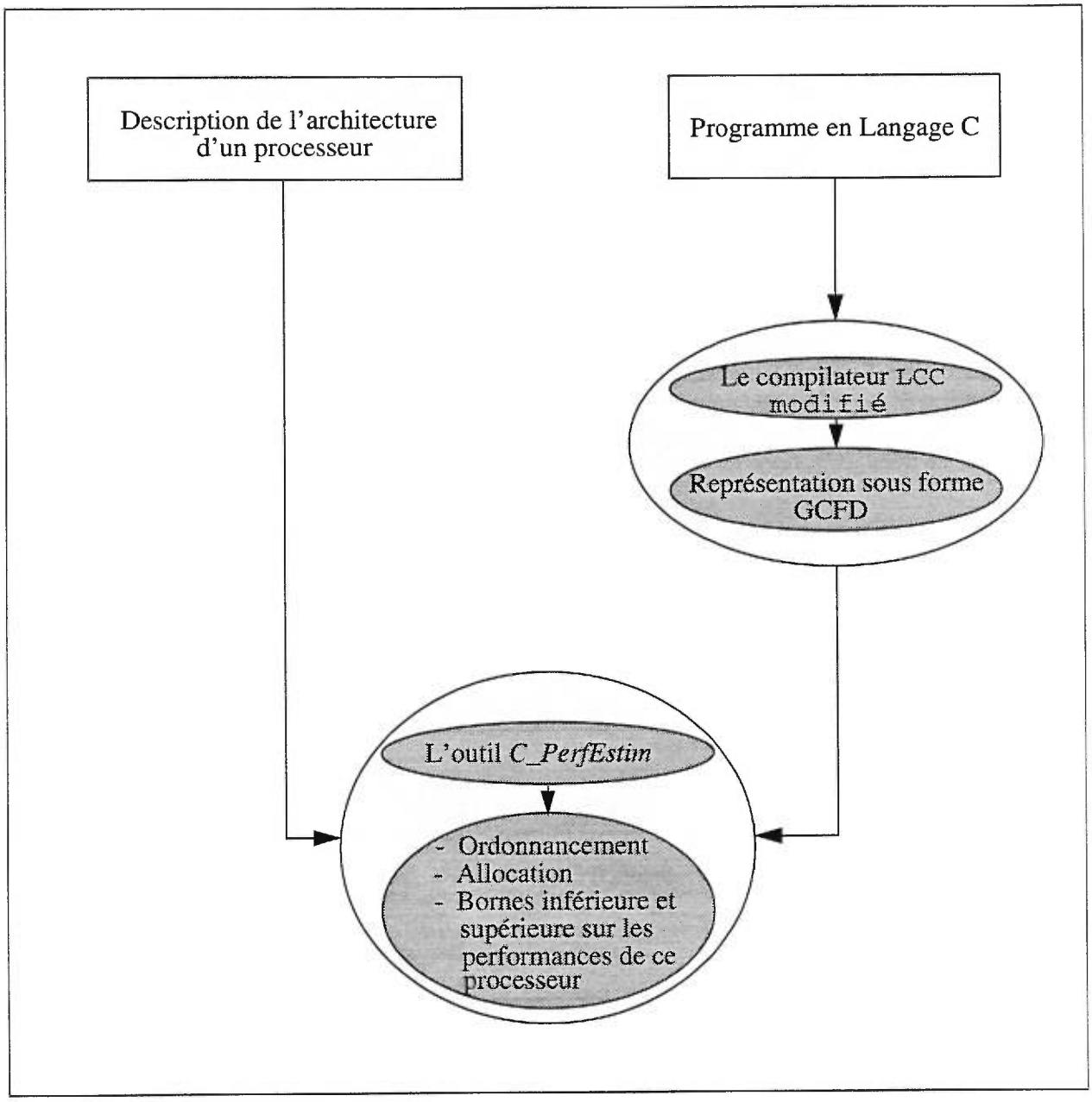


Figure 5.2 Vue d'ensemble de l'outil C_PerfEstim.

5.2.1- Description de l'architecture du processeur pour l'outil *C_PerfEstim*

Dans l'outil *C_PerfEstim*, la description de l'architecture du processeur est donnée par un ensemble de cinq fichiers comme il est montré à la Figure 5.3. Ces fichiers sont *op_latence.lib*, *bus_sources_font.lib*, *bus_destin_font.lib*, *ressources_infos.lib* et *ressources_font.lib*. Nous expliquons le contenu de chacun de ces fichiers dans les prochains paragraphes.

Pour ajouter des commentaires aux données d'un fichier, plusieurs façons s'offrent. Pour chacun des cinq fichiers, nous avons choisi l'ajout de commentaires à la fin des données. Pour marquer la fin de la lecture de ces données (i.e., pour séparer la zone des données de la zone des commentaires), le besoin d'un indicateur s'impose. Dans notre cas, nous avons choisi -1 comme indicateur de la fin de la lecture des données de chacun de ces fichiers, puisque -1 n'apparaît pas dans celles-ci. Nous continuons l'explication du contenu de chacun de ces fichiers dans les paragraphes suivants.

Le fichier *op_latence.lib* contient la liste des opérateurs supportés par le compilateur *LCC* ainsi que leurs latences. Ces opérateurs sont présentés au Chapitre 3. Nous ajoutons à cette liste l'opérateur *nop* "No Operation". Cet opérateur constitue le contenu de chacun des noeuds de synchronisation dans les blocs de base du graphe de contrôle et de flot de données.

Le fichier *bus_sources_font.lib* contient le nombre de bus sources et les opérateurs qui requièrent l'unité de stockage en lecture. Nous exploitons le contenu de ce fichier pour la gestion des connexions pour ce type de bus.

Le fichier *bus_destin.lib* contient le nombre de bus destinations et les opérateurs qui requièrent des unités fonctionnelles. Nous exploitons le contenu de ce fichier pour la gestion des connexions pour ce type de bus.

Le contenu du fichier *ressources_infos.lib* est une description des ressources du processeur pour lequel on veut estimer les performances. Ce fichier contient le nombre de types différents de ressources et une série d'éléments. Chaque élément de cette série a:

- un nom qui est le nom de la famille de ressources d'un même type;
- un nombre qui est le nombre d'exemplaires de cette famille de ressources;

- une contrainte (voir un exemple plus loin) qui vaut zéro s'il n'y a pas de contrainte sur ce type de ressources, autrement, la valeur de la contrainte;
- un style qui vaut un si les ressources de cette famille sont de type "pipeline", autrement, ce style vaut zéro;
- et finalement, un intervalle d'initialisation.

Quand les ressources d'un même type ne sont pas de type "pipeline", l'intervalle d'initialisation est la latence d'une de celles-ci. La latence d'une ressource est l'intervalle du temps minimal qui sépare le temps d'ordonnancement de deux opérations indépendantes sur celle-ci.

Le fichier *ressources_font.lib* contient le nombre de types différents de ressources du processeur pour lequel on veut estimer les performances et une série d'éléments. Chaque élément de cette série se caractérise par un nom qui est le nom des ressources d'un même type, par la liste d'opérateurs supportés par ce type de ressources, et par la chaîne de caractères _Fin_. La chaîne de caractères _Fin_ nous permet de séparer la liste des opérateurs supportés par chaque type de ressource.

oper_latence.lib

```
Opérateur       sa latence en nombre de cycles d'horloge
:                    :
-1 /* -1: signifie la fin du fichier */
/* Ci-dessous est un exemple*/
nop             0
ADD             2
MUL             2
-1
```

bus_sources_font.lib

```
Nombre de bus sources
      Noms des opérateurs qui requièrent
      des unités alimentant ce type de bus.
-1 /* -1: signifie fin du fichier */
/* Ci-dessous est un exemple */
4
  INDIR
  CNST
-1
```

bus_destin_font.lib

```
Nombre de bus destinations
      Noms des opérateurs qui requièrent
      des unités fonctionnelles.
-1 /* -1: signifie fin du fichier */
/* Ci-dessous est un exemple */
4
    ADD
    SUB
-1
```

ressources_infos.lib

```
Nombre de types différents des ressources
Nom d'unités de même type
      Nombre d'exemplaires de ces unités
      Contrainte? (0/valeur de la contrainte)
      Style? 1 en cas pipeline, 0 sinon
      Intervalle d'initialisation
.
.
.
Nom d'unités de même type
      Nombre d'exemplaires de ces unités
      Contrainte? (0/valeur de la contrainte)
      Style? 1 en cas pipeline, 0 sinon
      Intervalle d'initialisation
-1 /* -1: signifie fin du fichier */
/* Ci-dessous est un exemple */
1
ALU  1  0  1  1
-1
```

ressources_font.lib

```
Nombre de types differents des ressources
Nom d'unités de même type
      Noms des opérateurs supportés
_Fin_
.
.
.
Nom d'unités de même type
      Noms des opérateurs supportés
_Fin_
-1 /* -1 signifie fin du fichier */
/* Ci-dessous est un exemple */
1
ALU
    ADD
    SUB
_Fin_
-1
```

Figure 5.3 Fichiers de description de l'architecture d'un processeur dans l'outil C_PerfEstim.

5.2.2- Le graphe de contrôle et de flot de données: GCFD

Comme nous l'avons vu au Chapitre 2, le GCFD est une représentation intermédiaire d'un programme décrivant la tâche à réaliser par un système donné. Dans l'outil *C_PerfEstim*, le GCFD s'obtient par la traduction d'un programme écrit en Langage C. Cette traduction se fait à l'aide du compilateur *LCC* modifié. Nous avons examiné les modifications apportées à ce compilateur au Chapitre 3. La Figure 5.4 présente un exemple de programme dont le graphe GCFD alimente l'outil *C_PerfEstim;* nous revenons à cet exemple à la Section 5.4. Nous pouvons visualiser graphiquement ce CGFD à l'aide de l'outil *VCG* que nous avons intégré dans le compilateur *LCC* modifié.

```
main()
{
        int i, n;
        int a[5], x[256], y[256];

        for (/*$ 256 256 */ n=0; n<256; n++)
            for (/*$ 5 5  */ i=0; n<5; i++) {
                    y[n] = y[n] + a[i]*x[n-i];
            }
}
```

Figure 5.4 Exemple de programmes dont le graphe GCFD alimente l'outil C_PerfEstim.

5.3- Quelques applications du domaine du traitement d'images

Le filtrage est une des applications les plus fréquemment utilisées dans le domaine du traitement d'images [Maci93]. Il en existe plusieurs types comme les filtres à réponse impulsionnelle finie unidimensionnelle (RIF 1-D), les filtres à réponse impulsionnelle finie bidimensionnelle (RIF 2-D), et les filtres à réponse impulsionnelle infinie (RII 1-D) unidimensionnelle. Nous présentons les équations qui définissent les traitements réalisés par ces filtres dans ce qui suit.

5.3.1- Filtre RIF 1-D

Il s'agit de calculer un ensemble d'éléments y_n définis par:

$$y_n = \sum_{i=0}^{K-1} a_i \cdot x_{n-i}$$

où K est l'ordre du filtre RIF 1-D et, a_i et x_{n-i} sont, respectivement, ses coefficients et ses entrées.

5.3.2- Filtre RIF 2-D

Il s'agit de calculer un ensemble d'éléments $y_{m,n}$ définis par:

$$y_{m,n} = \sum_{i=0}^{H-1} \sum_{j=0}^{K-1} a_{i,j} \cdot x_{m-i,n-j}$$

où H et K sont les ordres du filtre RIF 2-D et, $a_{i,j}$ et $x_{m-i,n-j}$ sont, respectivement, ses coefficients et ses entrées. Dans ce mémoire, nous supposons que H et K sont égaux.

5.3.3- Filtre RII 1-D

Il s'agit de calculer un ensemble d'éléments y_n définis par:

$$y_n = \sum_{i=0}^{K-1} a_i \cdot x_{n-i} - \sum_{i=1}^{K-1} b_i \cdot y_{n-i}$$

où K est l'ordre du filtre RII 1-D, a_i et b_i sont ses coefficients, et x_{n-i} sont ses entrées.

5.4- Application de l'outil *C_PerfEstim* à l'estimation de performances de la machine *PULSE V1*

PULSE V1 est une machine parallèle de type SIMD. Elle est constituée de quatre processeurs parallèles de type "Very Long Instruction Word" (VLIW). Ils sont identiques. Cette machine est optimisée pour le traitement d'images [MarK98]. Pour plus de détails sur la machine *PULSE V1*, le lecteur peut consulter la documentation technique de celle-ci à la référence [PULSEV1].

Nous consacrons cette section à l'estimation du temps de calcul des filtres RIF 1-D, RIF 2-D, et RII 1-D sur la machine *PULSE V1* en utilisant l'outil *C_PerfEstim*. La Figure 5.5 (a) présente

un code en Langage C correspondant au calcul de 2^{10} éléments de sortie du filtre RIF 1-D d'ordre cinq. Dans ce code, il n'y a pas de dépendances inter-itérations au niveau de la boucle externe. Par conséquent, cette dernière est parallélisable. La Figure 5.5 (b) présente le code à exécuter sur un des processeurs de la machine *PULSE V1*. Nous obtenons ce code par un partitionnement de celui de la Figure 5.5 (a) sur les quatre processeurs de cette machine.

(a) Code en Langage C correspondant au calcul de 2^{10} éléments de sortie d'un filtre RIF 1-D d'ordre cinq.

```
main ()
{
  int i, n;
  int a[5], x[1024], y[1024];

  for (n=0; n<1024; n++)
    for (i=0; i<5; i++)
      y[n] += a[i] * x[n-i];
}
```

(b) Code en Langage C correspondant au calcul de 2^{8} éléments de sortie d'un filtre RIF 1-D d'ordre cinq sur un des processeurs de la machine *PULSE V1*.

```
main ()
{
  int i, n;
  int a[5], x[256], y[256];

  for (n=0; n<256; n++)
    for (i=0; i<5; i++)
      y[n] += a[i] * x[n-i];
}
```

Figure 5.5 Codes en Langage C correspondant au calcul d'un ensemble d'éléments de sortie d'un filtre RIF 1-D d'ordre cinq.

La Figure 5.6 (a) présente un code en Langage C correspondant au calcul de 2^{20} éléments de sortie du filtre RIF 2-D d'ordre cinq. Dans ce code, il n'y a pas de dépendances inter-itérations au niveau de la boucle externe. En conséquence, cette dernière est parallélisable. La Figure 5.6 (b) présente le code à exécuter sur un des processeurs de la machine *PULSE V1*. Nous obtenons ce

code par un partitionnement de celui de la Figure 5.6 (a) sur les quatre processeurs de cette machine.

(a) Code en Langage C correspondant au calcul de 2^{20} éléments de sortie d'un filtre RIF 2-D d'ordre cinq.

```
main ()
{
  int  i, j, m, n;
  int  a[5][5], x[1024][1024], y[1024][1024];

  for (m=0; m<1024; m++)
    for (n=0; n<1024; n++)
      for (i=0; i<5; i++)
        for (j=0; j<5; j++)
          y[m][n] += a[i][j] * x[m-i][n-j];
}
```

(b) Code en Langage C correspondant au calcul de 2^{18} éléments de sortie d'un filtre RIF 2-D d'ordre cinq sur un des processeurs de la machine *PULSE V1*.

```
main ()
{
  int  i, j, m, n;
  int  a[5][5], x[256][1024], y[256][1024];

  for (m=0; m<256; m++)
    for (n=0; n<1024; n++)
      for (i=0; i<5; i++)
        for (j=0; j<5; j++)
          y[m][n] += a[i][j] * x[m-i][n-j];
}
```

Figure 5.6 Codes en Langage C correspondant au calcul d'un ensemble d'éléments de sortie d'un filtre RIF 2-D d'ordre cinq.

La Figure 5.7 (a) présente un code en Langage C correspondant au calcul de 2^{10} éléments de sortie du filtre RII 1-D d'ordre cinq. Dans ce code, la référence aux éléments `y[n]` et `y[n-i]` induit des dépendances inter-itérations au niveau de la boucle externe. Par conséquent, le calcul parallèle de ces éléments sur plusieurs processeurs requiert des communications inter-processeurs. La boucle la plus interne de ce code est parallélisable, mais son partitionnement sur plusieurs processeurs ne permet pas de calculer en parallèle des éléments de sortie de ce filtre. Nous

rappelons qu'à cause que nous ne disposons pas de compilateur parallèle, l'outil *C_PerfEstim* ne supporte pas des programmes contenant de la communication inter-processeurs. Puisque nous nous intéressons à l'estimation du temps d'exécution de ce code sur la machine *PULSE V1* en utilisant cet outil, nous supposons dans ce cas qu'un seul processeur va exécuter le code de la Figure 5.7 (a).

Maintenant que nous avons discuté du code à exécuter sur les processeurs de la machine *PULSE V1*, nous examinons dans la suite la description de l'architecture d'un processeur de cette machine dans l'outil *C_PerfEstim*; nous rappelons que les processeurs de cette machine sont identiques. Ensuite, nous présentons les résultats d'estimation du temps d'exécution des codes présentés dans les Figures 5.5 (a), 5.6 (a) et 5.7 (a) sur cette machine.

(**a**) Code en Langage C correspondant au calcul de 2^{10} éléments de sortie d'un filtre RII 1-D d'ordre cinq.

```
main ()
{
  int i, n;
  int a[5], b[5], x[1024], y[1024];

  for (n=0; n<1024; n++)
  {
    y[n] = a[0] * x[0];
    for (i=1; i<5; i++)
      y[n] += a[i] * x[n-i] - b[i] * y[n-i];
  }
}
```

Figure 5.7 Code en Langage C correspondant au calcul d'un ensemble d'éléments de sortie d'un filtre RII 1-D d'ordre cinq.

5.4.1- Description de l'architecture d'un processeur de la machine *PULSE V1* dans l'outil *C_PerfEstim*

Pour faciliter l'explication des fichiers de description de l'architecture d'un processeur de la machine *PULSE V1* dans l'outil *C_PerfEstim*, nous présentons le modèle simplifié de

l'architecture de ce processeur dans la Figure 5.8. Ce dernier est constitué d'une mémoire, d'un générateur de constantes, d'un accumulateur, de quatre bus source, d'un multiplicateur-additionneur, d'un décaleur, d'une unité arithmétique et logique ainsi que de quatre bus destinations. Comme il est illustré dans la Figure 5.8, nous avons ajouté des blocs appelés *connexions dynamiques* pour connecter les bus avec les autres composants du processeur; ceci est pour éviter l'encombrement de cette figure.

La Figure 5.9 présente les cinq fichiers de description de l'architecture d'un processeur de la machine *PULSE V1*. Ces fichiers sont: *op_latence.lib*, *bus_sources_font*, *bus_destin_font.lib*, *ressources_infos.lib* et *ressources_font.lib*. Nous expliquons le contenu de ces fichiers dans les paragraphes suivants.

Le fichier *op_latence.lib* contient tous les opérateurs qui peuvent constituer le contenu des noeuds des blocs de base du graphe de contrôle et de flot de données correspondant au programme à exécuter sur un processeur de la machine *PULSE V1*. Ce fichier contient aussi la durée, en terme de cycles d'horloge, de l'exécution de l'opération correspondante à un de ces opérateurs sur une ressource de ce processeur. Par exemple, l'exécution de l'opération correspondante à l'opérateur *nop* requiert 0 cycles.

Le fichier *bus_sources_font.lib* contient le nombre de bus source d'un processeur de la machine *PULSE V1*. Dans ce cas, nous avons quatre bus sources. Ce fichier contient aussi l'ensemble des opérateurs qui requièrent un port mémoire en lecture, le générateur de constantes ou l'accumulateur. Dans ce cas, ces opérateurs sont *INDIR* et *CNST*. Nous utilisons le contenu de ce fichier pour gérer les connexions pour les bus source.

Le fichier *bus_destin_font.lib* contient le nombre de bus destinations d'un processeur de la machine *PULSE V1*. Dans ce cas, nous avons quatre bus destinations. Ce fichier contient aussi l'ensemble des opérateurs qui requièrent le multiplicateur-additionneur, le décaleur ou l'unité arithmétique et logique. Par exemple, l'opérateur *BCOM* requiert l'unité arithmétique et logique. Nous utilisons le contenu de ce fichier pour gérer les connexions pour les bus destination.

Le fichier *ressources_infos.lib* contient le nombre de types différents de ressources d'un

processeur de la machine *PULSE V1* ainsi que des informations sur celles-ci. Dans ce cas, nous avons neuf types de ressources:

- *Ports_mem_l*: Ports mémoires en lecture.
- *Gener_const*: Générateur de constantes.
- *Acc*: Accumulateur.
- *Bus_source*: Bus source.
- *MulAdd*: Multiplicateur-additionneur.
- *Decaleur*: Décaleur.
- *UAL*: Unité arithmétique et logique.
- *Bus_destin*: Bus destination.
- *Ports_mem_e*: Ports mémoires en écriture.

Les informations sur chaque type de ressources sont: le nombre d'exemplaires de ce type de ressources, présence ou absence de contraintes sur celles-ci, le style de ce type de ressources ainsi que l'intervalle d'initialisation de celles-ci. Par exemple, pour la deuxième ligne de ce fichier, nous avons: *Ports_mem_l* 4 6 0 1. Le nombre 4 signifie que nous avons quatre ports mémoires en lecture. Le nombre 6 signifie que, pour un instant donné, la somme du nombre des ports mémoires en lecture et du nombre des ports mémoires en écriture ne doit pas dépasser six. Le nombre 0 signifie que les quatre ports mémoires en lecture ne sont pas de type "pipeline". Le nombre 1 signifie que chacun des quatre ports mémoires en lecture peut être immédiatement utilisé après un cycle de son utilisation.

Le fichier ressources_font.lib contient le nombre de types différents de ressources d'un processeur de la machine *PULSE V1* ainsi que les opérateurs supportés par les ressources de même type (e.g., *INDIR* est un opérateur supporté par *Ports_mem_l*). Dans ce cas, nous avons neuf types de ressources. Ces derniers sont les mêmes que ceux du fichier *ressources_infos.lib*.

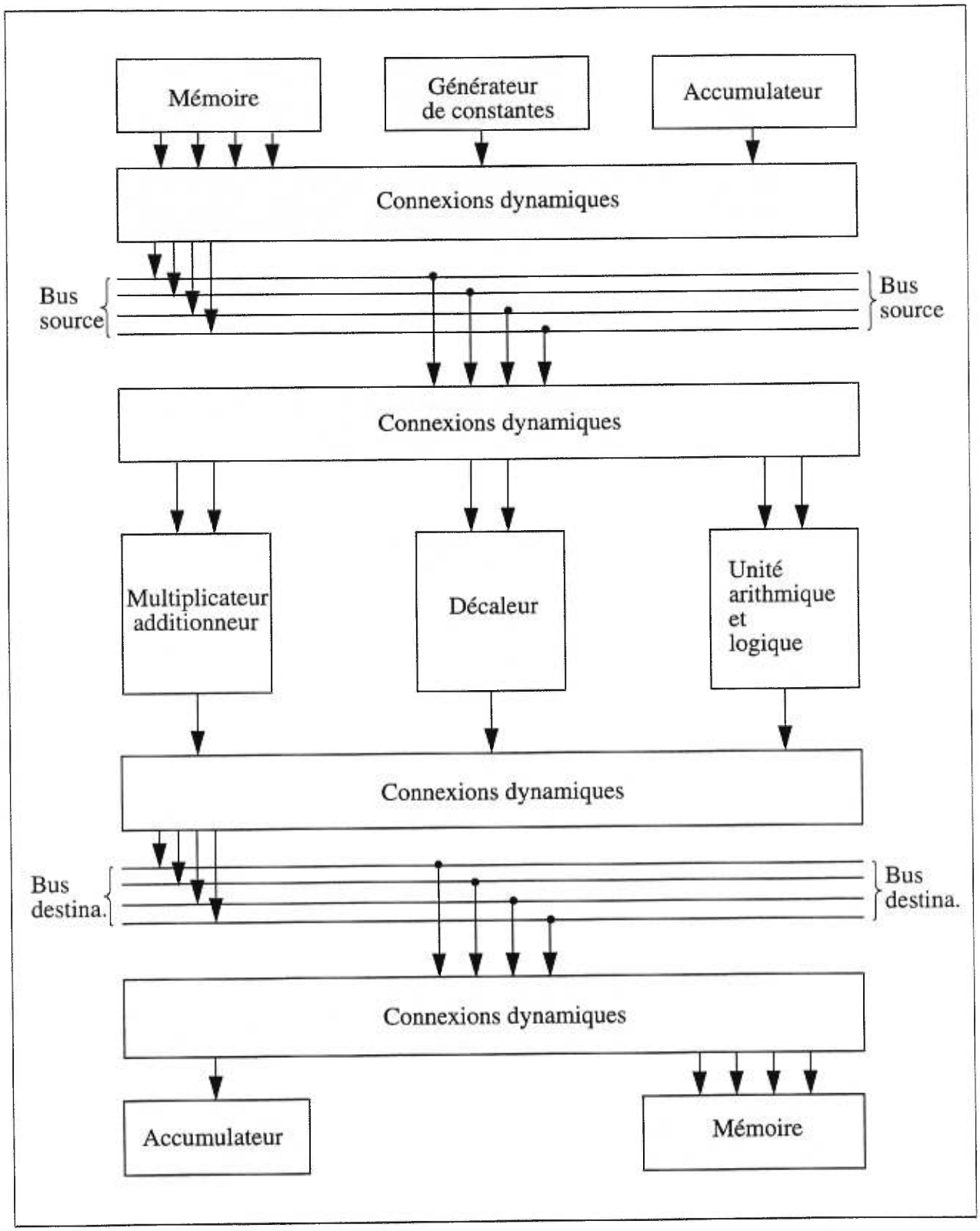


Figure 5.8 Modèle simplifié de l'architecture d'un processeur de la machine PULSE V1.

op_latence.lib

```
nop     0
ADDRT   2
ADDRF   2
ADDRG   2
ADDRL   2
CNST    1
BCOM    2
CVC     2
CVD     2
CVF     2
CVI     2
CVP     2
CVS     2
CVU     2
INDIR   1
NEG     2
ADD     2
BAND    2
BOR     2
BXOR    2
DIV     2
LSH     2
MOD     2
MUL     2
RSH     2
SUB     2
ASGN    1
EQ      2
GE      2
GT      2
LE      2
LT      2
NE      2
ARG     2
CALL    2
RET     2
JUMP    2
LABEL   2
-1
```

bus_sources_font.lib

```
4
    INDIR
    CNST
-1
```

bus_destin_font.lib

```
4
    BCOM
    CVC
    CVD
    CVF
    CVI
    CVP
    CVS
    CVU
    NEG
    ADD
    BAND
    BOR
    BXOR
    DIV
    LSH
    MOD
    MUL
    RSH
    SUB
    EQ
    GE
    GT
    LE
    LT
    NE
    JUMP
    LABEL
-1
```

ressources_infos.lib

```
9
    Ports_mem_l     4   6   0   1
    Gener_const     1   0   0   1
    Acc             1   0   0   1
    Bus_source      4   0   0   1
    MulAdd          1   0   1   1
    Decaleur        1   0   1   1
    UAL             1   0   1   1
    Bus_destin      4   0   0   1
    Ports_mem_e     4   6   0   1
-1
```

ressources_font.lib

```
9
Ports_mem_l
    INDIR
_Fin_
Gener_const
    CNST
_Fin_
Acc
_Fin_
Bus_source
_Fin_
MulAdd
    MUL
    ADD
_Fin_
Decaleur
    RSH
    LSH
_Fin_
UAL
    BCOM
    CVC
    CVD
    CVF
    CVI
    CVP
```

ressources_font.lib (suite)

```
    CVS
    NEG
    CV
    ADD
    BAND
    BOR
    BXOR
    DIV
    MOD
    SUB
_Fin_
Bus_destin
_Fin_
Ports_mem_e
    ASGN
_Fin_
-1
```

Figure 5.9 Fichiers de description de l'architecture d'un processeur de la machine PULSE V1.

5.4.2- Résultats expérimentaux et discussions

Cette section présente les résultats de l'estimation du temps de calcul d'un ensemble d'éléments de sortie des filtres RIF 1-D, RIF 2-D et RII 1-D sur la machine *PULSE VI* en utilisant l'outil *C_PerfEstim*. Ces filtres sont d'ordre cinq dans chaque dimension. Cette section présente aussi des discussions de ces résultats.

La Figure 5.10 présente l'estimation du temps de calcul de 2^{10} éléments de sortie du filtre RIF 1-D sur la machine *PULSE VI*. Plus précisément, cette figure présente le code à exécuter sur un des processeurs de cette machine, le graphe de contrôle et de flot de données (GCFD) correspondant à ce code, les performances des blocs de base de ce GCFD ainsi que les performances de celui-ci. Dans le code de la Figure 5.10 (a), le commentaire spécial `/*$ 256 256 */` signifie que chacun des nombres minimal et maximal d'itérations de la boucle vaut 256. Ces nombres aident à la détermination d'une borne inférieure et d'une borne supérieure sur le temps d'exécution de ce GCFD. Pour chaque bloc de base (BB) de ce GCFD, la Figure 5.10 (c) présente la longueur du chemin critique dans ce BB, une borne inférieure et une borne supérieure sur le temps d'exécution de celui-ci, ainsi que l'écart relatif de ces deux bornes. La longueur du chemin critique d'un BB est la valeur du plus long chemin dans celui-ci. Nous définissons l'écart relatif des deux bornes inférieure et supérieure comme suit:

$$\textit{Écart relatif} = (\textit{Borne supérieure} - \textit{Borne inférieure}) / \textit{Borne supérieure}\ .$$

La Figure 5.10 (d) présente une borne inférieure et une borne supérieure sur le temps d'exécution de ce GCFD ainsi que l'écart relatif de ces deux bornes.

La Figure 5.11 présente l'estimation du temps de calcul de 2^{20} éléments de sortie du filtre RIF 2-D sur la machine *PULSE VI*. L'estimation du temps de calcul de 2^{10} éléments de sortie du filtre RII 1-D sur cette machine est présentée à la Figure 5.12. L'explication des composants de ces deux figures est similaire à celle de la Figure 5.10.

Nous rappelons que la longueur de chemin critique dans un bloc de base (BB) est la longueur de plus long chemin dans ce BB. Quand il n'y a pas de contraintes de ressources du processeur pour lequel on veut estimer les performances, la longueur du chemin critique d'un bloc de base représente la borne inférieure sur le temps d'exécution de celui-ci.

Quand on a une contrainte de ressources, la borne inférieure (*Binf*) sur le temps d'exécution d'un bloc de base (BB) est supérieure ou égale à la longueur du chemin critique. Les résultats obtenus aux Figures 5.10 (a), 5.11 (a), et 5.12 (a) montrent que la différence entre la longueur du chemin critique et une borne inférieure sur le temps d'exécution pour chaque bloc de base est dans l'intervalle [0, 1]. Cette faible différence nous assure que l'architecture de chacun des processeurs de la machine *PULSE V1* n'est pas une contrainte forte sur le temps d'exécution de ces trois applications (i.e., calcul d'un ensemble d'éléments sorties des filtres RIF 1-D, RIF 2-D et RII 1-D). En conséquence, l'architecture de ces processeurs ne limite pas l'exploitation du parallélisme existant dans ces applications.

Nous définissons le temps d'exécution optimal d'un bloc de base (BB) comme étant le minimum de temps requis pour l'ordonnancement et l'allocation de tous les noeuds de ce BB sur l'architecture du processeur pour lequel nous voulons estimer les performances, toute en considérant les trois contraintes suivantes: (1) la contrainte des dépendances de données, (2) la contrainte de ressources et (3) la contrainte des interconnexions.

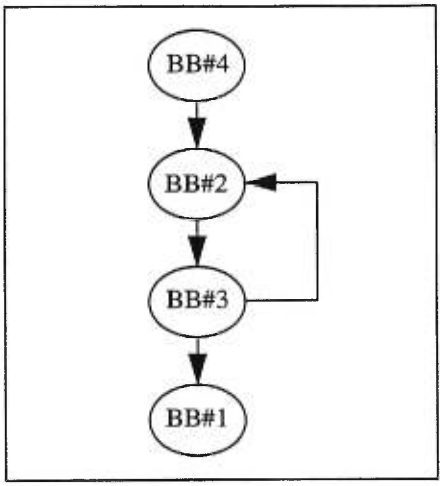
Une borne supérieure sur le temps d'exécution (*Bsup*) d'un BB est le temps nécessaire pour l'exécution de tous ses noeuds. Nous obtenons une borne *Bsup* sur le temps d'exécution d'un BB en ordonnançant et en allouant les noeuds de celui-ci sur l'architecture du processeur pour lequel nous voulons estimer les performances sous les trois contraintes ci-dessus. Le temps d'exécution optimal de ce BB est dans l'intervalle [*Binf, Bsup*]. Afin de l'atteindre, il faut que l'écart relatif des deux bornes inférieure et supérieure sur le temps d'exécution de ce BB soit nul; ceci n'est pas toujours possible car le problème d'ordonnancement et d'allocation est un problème NP-complet. Les résultats obtenus aux Figures 5.10 (a), 5.11 (a), et 5.12 (a) montrent que l'écart relatif des deux bornes inférieure et supérieure sur le temps d'exécution de chaque bloc de base est dans l'intervalle [0%, 7.1%]. Les faibles valeurs de cet écart montrent l'efficacité des algorithmes sur lesquels se base l'outil *C_PerfEstim*.

En conclusion, l'architecture des processeurs de la machine *PULSE V1* est convenable pour l'accélération du traitement de ces applications (i.e., calcul d'un ensemble d'éléments de sortie des filtres RIF 1-D, RIF 2-D et RII 1-D).

```
main ()
{
  int n, s;
  int a[5], x[256], y[256];

  for (/*$ 256 256 */n=0; n<256; n++)
  {
    s  = a[0] * x[n];
    s += a[1] * x[n-1];
    s += a[2] * x[n-2];
    s += a[3] * x[n-3];
    s += a[4] * x[n-4];
    y[n] = s;
  }
```

(a) Code en Langage C correspondant au calcul de 2^8 éléments de sortie d'un filtre RIF 1-D d'ordre cinq sur un processeur de la machine *PULSE V1*.



(b) Graphe de contrôle et de flot de données (GCFD) correspondant à (a).

Blocs de base (BBs)	Longueur du chemin critique dans un BB (en cycles d'horloge)	Borne inférieure sur le temps d'exéc. d'un BB (en cycles d'horloge)	Borne supérieure sur le temps d'exéc. d'un BB (en cycles d'horloge)	Écart relatif des deux bornes inférieure et supérieure (%)
BB#4	4	4	4	0
BB#2	36	37	37	0
BB#3	7	7	7	0
BB#1	∞	0	0	Indéfini

(c) Performances des blocs de base du graphe GCFD correspondant à (a).

Borne inférieure sur le temps d'exécution du GCFD (en cycles d'horloge)	Borne supérieure sur le temps d'exécution du GCFD (en cycles d'horloge)	Écart relatif des deux bornes inférieure et supérieure (%)
$11.26\ 10^4$	$11.26\ 10^4$	0

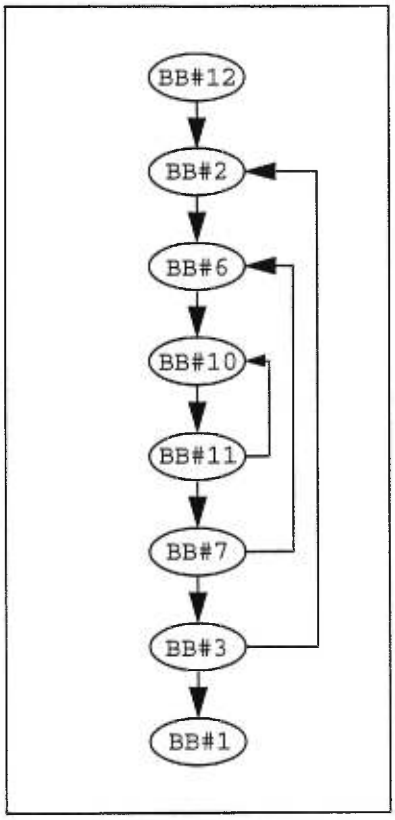
(d) Performances du graphe GCFD correspondant à (a).

Figure 5.10 Estimation du temps de calcul de 2^{10} éléments de sortie d'un filtre RIF 1-D d'ordre cinq sur la machine PULSE V1.

```
main ()
{ int  i, m, n, y1, y2, y3, y4, y5;
  int  a[5][5], x[256][1024], y[256][1024];

  for (/*$ 256 256*/ m=0; m<256; m++){
    for (/*$ 1024 1024*/ n=0; n<1024; n++){
      for (/*$ 5 5*/ i=0; i<5; i++) {
        y1 = a[i][0] * x[m-i][n];
        y2 = a[i][1] * x[m-i][n-1];
        y3 = a[i][2] * x[m-i][n-2];
        y4 = a[i][3] * x[m-i][n-3];
        y5 = a[i][4] * x[m-i][n-4];
        y[m][n] += y1 + y2 + y3 + y4 + y5;
      }
    }
  }
}
```

(a) Code en Langage C correspondant au calcul de 2^{18} éléments de sortie d'un filtre RIF 2-D d'ordre cinq sur un processeur de la machine *PULSE V1*.



(b) Graphe de contrôle et de flot de données (GCFD) correspondant à (a).

Blocs de base (BBs)	Longueur du chemin critique dans un BB (en cycles d'horloge)	Borne inf. sur le temps d'exéc. d'un BB (en cycles d'horlog.)	Borne sup. sur le temps d'exéc. d'un BB (en cycles d'horlog.)	Écart relatif des deux bornes inférieure et supérieure (%)
BB#12	4	4	4	0
BB#2	4	4	4	0
BB#6	4	4	4	0
BB#10	58	59	63	6.3
BB#11	7	7	7	0
BB#7	7	7	7	0
BB#3	7	7	7	0
BB#1	∞	0	0	Indéfini

(c) Performances des blocs de base correspondant au graphe de contrôle et de flot de données de (a).

Borne inf. sur le temps d'exéc. du GCFD (en cycles d'horloge)	Borne sup. sur le temps d'exéc. du GCFD (en cycles d'horloge)	Écart relatif des deux bornes inférieure et supérieure (%)
$8.93\ 10^7$	$9.46\ 10^7$	5.6

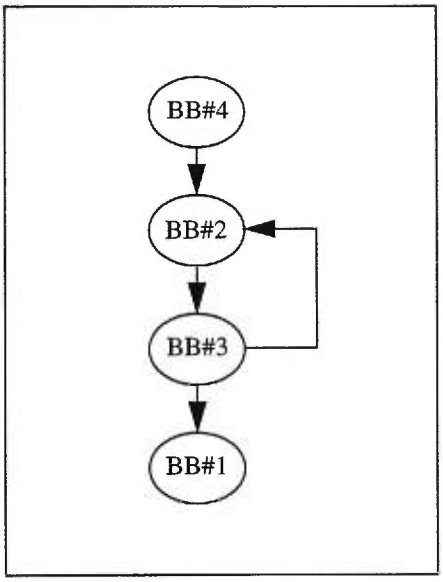
(d) Performances du graphe de contrôle et de flot de données de (a).

Figure 5.11 Estimation du temps de calcul de 2^{20} éléments de sortie d'un filtre RIF 2-D d'ordre cinq sur la machine PULSE V1.

```
main ()
{ int n, sa, sb;
  int a[5], b[5], x[1024], y[1024];

  for (/*$ 1024 1024 */n=0; n<1024; n++)
  { sa  = a[0] * x[n];
    sa += a[1] * x[n-1];
    sa += a[2] * x[n-2];
    sa += a[3] * x[n-3];
    sa += a[4] * x[n-4];
    sb  = b[1] * y[n-1];
    sb += b[2] * y[n-2];
    sb += b[3] * y[n-3];
    sb += b[4] * y[n-4];
    y[n] = sa - sb;
  }
}
```

(a) Code en Langage C correspondant au calcul de 2^{10} éléments de sortie d'un filtre RII 1-D d'ordre cinq sur un processeur de la machine *PULSE V1*.



(b) Graphe de contrôle et de flot de données (GCFD) correspondant à (a).

Blocs de base (BBs)	Longueur du chemin critique dans un BB (en cycles d'horloge)	Borne inférieure sur le temps d'exécution d'un BB (en cycles d'horloge)	Borne supérieure sur le temps d'exécution d'un BB (en cycles d'horloge)	Écart relatif des deux bornes inférieure et supérieure (%)
BB#4	4	4	4	0
BB#2	38	39	42	7.1
BB#3	7	7	7	0
BB#1	∞	0	0	Indéfini

(c) Performances des blocs de base du graphe GCFD correspondant à (a).

Borne inférieure sur le temps d'exécution du GCFD (en cycles d'horloge)	Borne supérieure sur le temps d'exécution du GCFD (en cycles d'horloge)	Écart relatif des deux bornes inférieure et supérieure (%)
$4.71\ 10^4$	$5.01\ 10^4$	5.9

(d) Performances du graphe GCFD correspondant à (a).

Figure 5.12 Estimation du temps de calcul de 2^{10} éléments de sortie d'un filtre RII 1-D d'ordre cinq sur la machine PULSE V1.

Chapitre 6

Conclusion

Dans ce mémoire, nous nous somme intéressés à l'estimation des performances de systèmes informatiques parallèles de type "Single Instruction Stream over Multiple Data Streams" (SIMD). La motivation principale de ce travail est le besoin d'estimation des performances d'un ordinateur parallèle de type SIMD en cours de conception appelé *PULSE V1* destiné pour des applications réelles. Ce dernier se compose de quatre processeurs parallèles identiques de type "Very Long Instruction Word" (VLIW) ayant des ressources de type "pipeline". L'objectif de ce mémoire est de développer un outil informatique pour estimer les performances de systèmes parallèles de type SIMD et de l'appliquer à l'estimation des performances de l'ordinateur *PULSE V1*.

La programmation d'un système parallèle de type SIMD se caractérise par l'exécution d'un seul programme sur les processeurs de ce système sous le contrôle d'une seule unité de commande appelée contrôleur; tous les processeurs reçoivent la même instruction diffusée par le contrôleur, mais opèrent sur des données provenant de flux de données distincts. Par conséquent, en supposant que les processeurs de ce système sont identiques, l'estimation de ses performances revient à estimer les performances d'un de ses processeurs. Notons que ces derniers peuvent être des processeurs parallèles de type VLIW ayant des ressources de type "pipeline".

Pour estimer les performances d'un système, deux types de méthodes s'offrent: les méthodes basées sur la simulation et les méthodes statiques. Les méthodes basées sur la simulation ne s'appliquent pas tant qu'un modèle exécutable du système ne soit pas établi. Les méthodes statiques se basent sur une analyse globale de la structure du programme à exécuter sur le système. Elles peuvent être utilisées lors des premières phases de la conception du système ainsi qu'après sa fabrication.

Nous avons développé un outil informatique pour estimer les performances de systèmes parallèles de type SIMD. Cet outil est une implémentation d'une méthode statique d'estimation de performance. Cette méthode se base sur des algorithmes d'ordonnancement, d'allocation, et de calcul de plus court et de plus long chemins. Ensemble ces algorithmes permettent de déterminer des bornes inférieure et supérieure sur les performances. Certains de ces algorithmes sont des adaptations d'algorithmes existants dans la littérature, car ces derniers ont été conçus pour des architectures spécifiques.

Les algorithmes de cette méthode ont besoin d'un graphe de contrôle et de flot de données (GCFD) qui est une représentation intermédiaire d'un programme donné. Si le GCFD était généré pour une exécution séquentielle, des dépendances de données supplémentaires doivent être ajoutées. En plus, les informations suivantes doivent être présentes dans le GCFD quand ce dernier correspond à un traitement itératif. Ces informations sont le début et la fin de chaque boucle, ainsi que ses nombres minimal et maximal d'itérations. Il existe des compilateurs, tel que le compilateur *LCC*, qui génèrent un GCFD. Cependant, ils ne répondent pas aux besoins de ces algorithmes. Pour cela, nous avons apporté des modifications et des extensions au compilateur *LCC*. Afin de visualiser graphiquement le GCFD, nous avons implémenté une interface permettant de communiquer le compilateur *LCC* modifié avec un outil de visualisation graphique d'un graphe appelé *VCG*.

En général, le partitionnement d'un programme décrit dans un langage séquentiel en des sous-programmes dont l'exécution ne requiert pas de communications interprocesseurs peut ne pas être immédiat. Ceci découle du fait que le parallélisme dans ce programme peut être caché. Afin de faire apparaître ce parallélisme, des transformations de celui-ci sont requises. Pour cela, nous avons présenté un processus et des techniques de parallélisation automatique d'un programme. Ces techniques sont destinées à des traitements itératifs.

Nous avons utilisé l'outil développé pour estimer les performances de l'ordinateur *PULSE VI* en exécutant certaines applications du domaine de traitement d'images. Ces applications correspondent au calcul d'un ensemble d'éléments de sortie des filtres suivants: réponse impulsionnelle fini unidimensionnelle, réponse impulsionnelle fini bidimensionnelle, et réponse

impulsionnelle infini unidimensionnelle. Nous avons obtenu des résultats quasi-optimaux; nous entendons par quasi-optimaux le fait que l'écart relatif des bornes inférieures et supérieures sur les performances ainsi calculées (Section 5.4.2) est de valeurs faibles (i.e., ne dépassent pas 7.1%). Nous avons ainsi conclu que l'architecture des processeurs de cet ordinateur est convenable pour l'accélération du traitement de ces applications. Ces résultats nous ont aussi assuré l'efficacité des algorithmes sur lesquels se base l'outil développé.

Des avenues de recherches peuvent s'ouvrir comme suite logique à ce travail. Nous identifions les suivantes:

- Développement d'estimateurs de performances qui sont paramétrables (e.g., calcul symbolique des bornes inférieure et supérieures sur les performances d'un système).
- Développement d'une méthode statique pour estimer les ressources d'un système sous la contrainte de ses performances.

Bibliographie

[AllS96] James D. Allen, David E. Schimmel, "Issues in the Design of High Performance SIMD Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 8, August 1996.

[BalB94] N. D. Baltas, C. S. Van Den Berghe, "Comparison of The Porting of a Computational Fluid Dynamics Application to SIMD and MIMD Computers," *Massively Parallel Processing Applications and Development*, Elsevier Science, 1994.

[Bane88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic, 1988.

[BenL96] I. E. Bennour, M. Langevin, E. M. Aboulhamid, "Performance Analysis for Hardware/Software Co-synthesis," *Canadian Conference on Electrical and Computer Engineering*, pp. 162-165, *IEEE*, Calgary, AL, 1996.

[BerS91] Thomas B. Berg, Howard Jay Siegel, "Instruction Execution Trade-Offs for SIMD vs. MIMD vs. Mixed Mode Parallelism," *Proceedings of the* 5th *International Parallel Processing Symposium*, May 1991, pages 301-308.

[ChaA98] N. Chabini, E. M. Aboulhamid, Y. Savaria, I.E. Bennour,"A Static Method for System Performance Estimation," *The Tenth International Conference on Microelectronics*, 14-16 December 1998, Tunisia.

[Chré89] P. Chrétienne, "Task Scheduling Over Distributed Memory Machines," *Workshop on Parallel and Distributed Algorithms*, North-Holland, Amsterdam, 1989.

[ChrC95] P. Chrétienne, E. G. Coffman, J. J. K. Lenstra, Z. Liu, *Scheduling Theory and Its Applications*, John Wiley & Sons Ltd, 1995.

[CofD73] E. G. Coffman, P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N. J. , 1973.

[Dart93] A. Darte, *Techniques de parallélisation automatique de nids de boucles*, Thèse de doctorat, École Normale Supérieure de Lyon, Univ. Claude Bernard-Lyon 1, 1993.

[DeMi94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, Mc Graw-Hill, 1994.

[Dowl90] Michael. L. Dowling, "Optimal Code Parallelisation Using Unimodular Transformations," *Parallel Computing*, Elsevier Science Publishers B. V. North-Holland, 1990.

[Ferr78] D. Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Inc., 1978.

[FisH95] J. R. Fischer, L. E. Hamet, C. M. Mobarry, J. A. Pedelty, J. S. Cohen, R. K. de Fainchtein, B. A. Fryxell, P. J. MacNeice, K. M. Olson, T. L. Sterling, "The Practicality of SIMD for Scientific Computing," *The Fifth Symposium on The Frontiers of Massively Parallel Computation*, Feb. 6-9, 1995, McLean, Virginia.

[Flyn72] M. J., Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. c-21, No. 9, September 1972.

[FonD94] Cyril Fonlupt, Jean-Luc Dekeyser, Philippe Marquet, "Dynamic Load Balancing on SIMD Data-Parallel Computers," *Massively Parallel Processing Applications and Development*, Elsevier Science, 1994.

[FraH95] C. Fraser, D. Hanson, *A Retargetable C Compiler: Design and Implementation*, The Benjamin/Cummings Publishing Company, Inc, 1995.

[FraH91a] C. Fraser, D. Hanson, "A Retargetable Compiler for ANSI C," *SIGPLAN Notices* 26, 10 (Oct. 1991).

[FraH91b] C. Fraser, D. Hanson, "A Code Generation Interface for ANSI C," *Software-Practice and Experience* 21, 9 (Sep. 1991).

[Fren82] S. French, *Sequencing and Scheduling*, Halstead Press, New York, 1982.

[GajD92] D. D. Gajski, N. Dutt, A. Wu, S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, 1992.

[GajR94] D. D. Gajski, L. Ramachandran, "Introduction to High-Level Synthesis," *IEEE Design and Test of Computers*, 1994.

[GanG96] R. Ganesan, K. Govindarajan, M.Y. Wu, "Comparing SIMD and MIMD Programming Modes," *Journal of Parallel and Distributed Computing*, vol. 35, no. 1, pp. 91-96, May 1996.

[GonG94] J. Gong, D. D. Gajski, A. Nicolau, "A Performance Evaluator for Parametrized ASIC Architectures," *European Design Automation Conference*, 1994.

[Hwan84] K. Hwang, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984, pp. 355.

[Hwan93] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.

[IriT88] F. Irigoing, R. Triolet, *Computing Dependence Direction Vectors and Dependence Cones*, Rapport technique E94, Centre d'Automatique et Informatique, 1988.

[JainP92] R. Jain, A. C. Parker, N. Park, "Predicting System-Level Area and Delay for Pipelined and Non-Pipelined Designs," *IEEE Transactions on Computer-Aided-Design*, 11(8), August 1992.

[Kung88] H. T. Kung, "Warp Experience: We can map communications onto a parallel computer efficiently," *International Conference on SuperComputing Conference Proceedings*, St. Malo, France, July 1988.

[LCCV3.5] Code source du compilateur *LCC* V3.5, URL: `ftp://ftp.cs.princeton.edu/pub/packages/lcc`.

[Leig92] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures:*

Arrays.Trees. Hypercubes, Morgan Kaufman Publishers, 1992, pp. 60.

[LenR78] J. K. Lenstra, A. H. G. Rinnooy Kan, "Complexity of Scheduling Under Precedence Constraints," Oper. Res. 26, 1978.

[LinY97] Youn-Long Lin, "Recent Developments in High-Level Synthesis," *ACM Transactions on Design Automation of Electrical Systems*, Vol. 2, No. 1, January 1997.

[Maci93] Frederico Buchholz Maciel, *An Optimization-Aided Rate-Optimal Approach to the High-Level Synthesis of Digital Signal Processing Algorithms*, Doctoral Dissertation, Graduate School of Electronic Engineering, Faculty of Engineering, Hokkaido University, 1993.

[MarK98] P. Marriott, I. C. Kraljic, Y. Savaria, "Parallel Ultra Large Scale Engine SIMD Architecture For Real-Time Digital Signal Processing Applications," *International Conference on Computer Design*, October 5-7, 1998.

[MayH91] D. E. Mayadan, J. L. Hennessy, M. S. Lam, "Efficient and Exact Data Dependence Analysis," In Proc. *ACM Sigplan* 91 *Conference on Programming Language Design and Implementation*, June 1991.

[McFP90] M. C. McFarland, A. C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc. IEEE*, February 1990.

[Mead80] Carver Mead, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Inc, 1980, pp. 274.

[Moll89] Michael K. Molloy, *Fundamentals of Performance Modeling*, Macmillan Publishing Company, 1989, ISBN 0-02-381910-3.

[OhmJ92] S. Y. Ohm, C. S. Jhon, "A Branch-and-Bound Method for Optimal Scheduling," *Proceding of the 1992 Custom Integrated Circuits Conference, IEEE*, 1992.

[PULSEV1] Documentations techniques de la machine *PULSE V1*, Projet PULSE, École Polytechnique de l'Université de Montréal, Canada, 1997.

[RimJ96] M. Rim, R. Jain, "Valid Transformations: A New Class of Loop Transformations for High-Level Synthesis and Pipelined Scheduling Applications," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 4, April 1996.

[RimJ94] M. Rim, R. Jain, "Lower-bound Performance Estimation for High-Level Synthesis Scheduling Problem," *IEEE Transactions on CAD*, Vol. 3, pp. 81-88, 1994.

[Sand94] Peter Sander, "Emulating MIMD Behavior on SIMD Machines," *Massively Parallel Processing Applications and Development*, Elsevier Science, 1994.

[Sand95] G. Sander, "VCG: Visualization of Compiler Graphs," URL: `sander@cs.uni-sb.de`, Universität des Saarlandes, 66041 Saarbrücken Germany, Feb 9, 1995.

[Sark89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Executing on Multiprocessors*, MIT Press, Cambridge, MA, 1989.

[Sieg95] Howard Jay Siegel, "Panel on SIMD Machines: Do They Have a Significant Future?," *The Fifth Symposium on The Frontiers of Massively Parallel Computation*, 1995.

[TseW90] C.-W. Tseng, M. J. Wolfe, "The Power Test for Data Dependence," Technical Report Rice COMP TR90-145, Rice University, Dec. 1990..

[VCG1.3] Code source de l'outil VCG V1.30, URL: `ftp://ftp.cs.uni-sb.de/pub/graphics/vcg`.

[WalC95] R. A. Walker, S. Chaudhuri, "Introduction to the Scheduling Problem," *IEEE Design and Test of Computers*, 1995.

[Wolf92] M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*, Thèse de doctorat, Stanford University, 1992.

[WolL91] M. Wolf, M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel and Distributed Syst.*, Vol. 2, No.4, October 1991.

[WuSh96] Min-You Wu, Wei Shu, "MIMD Programs on SIMD Architectures," *The Sixth Symposium on The Frontiers of Massively Parallel Computation*, 1996.

[Zhen91] Si-Qing Zheng, "SIMD Data Communication Algorithms for Multiply Twisted Hypercubes," *Proceedings of the* 5th *International Parallel Processing Symposium*, May 1991, pages 120-125.