Université de Montréal

ESTIMATION DE LA PERFORMANCE ET MÉTHODES D'ALLOCATION DANS LA SYNTHÈSE DE SYSTÈMES NUMÉRIQUES

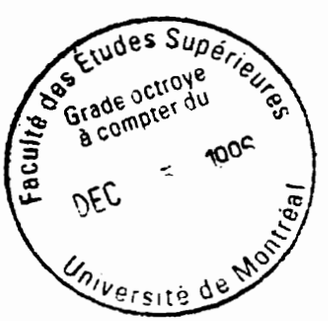
par

Imed-Eddine BENNOUR

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph. D.)
en informatique

Septembre, 1996



Université de Montréal
Faculté des études supérieures

Cette thèse intitulée

ESTIMATION DE LA PERFORMANCE ET MÉTHODES D'ALLOCATION DANS LA SYNTHÈSE DE SYSTÈMES NUMÉRIQUES

présentée par

Imed-Eddine BENNOUR

a été évaluée par un jury composé des personnes suivantes:

Xiaoyu SONG

(président-rapporteur)

El Mostapha ABOULHAMID

(directeur de recherche)

Yvon SAVARIA

(membre du jury)

Christian LEGER

(membre du jury)

Behrouz NOWROUZIAN

(examinateur externe)

Sommaire

La conception d'un circuit numérique s'effectue par des raffinements successifs de sa spécification jusqu'à l'obtention des masques de fabrication du silicium. Chaque raffinement réduit le niveau d'abstraction de la spécification en introduisant plus de détails. Au plus haut niveau d'abstraction, dit niveau comportemental, la spécification décrit uniquement la fonctionnalité du circuit en faisant abstraction de toute réalisation matérielle. La synthèse de haut-niveau (SHN) consiste à transformer une spécification d'un niveau comportemental à un niveau plus détaillé dit niveau transfert de registres ou RTL. Une description au niveau RTL est une description dans laquelle le circuit est représenté comme un assemblage d'éléments de mémorisation (registres) et d'unités de calcul et de contrôle. L'objectif de la SHN est de trouver la transformation qui donne la réalisation la plus performante et la moins coûteuse. Une telle transformation passe par la résolution de problèmes d'optimisation complexes tels que les problèmes d'ordonnancement et d'allocation.

Dans cette thèse nous présentons un ensemble de méthodes algorithmiques pour assister le processus de synthèse de haut-niveau. Nous proposons des méthodes pour calculer des estimateurs qui aideront le concepteur à évaluer plusieurs alternatives de réalisation d'un même circuit. Ces méthodes permettent de déterminer des bornes inférieures et supérieures sur la performance qu'une application donnée peut atteindre sur une architecture spécifique; des architectures séquentielles, parallèles et du type "pipeline" sont considérées. Tandis que les méthodes traditionnelles d'estimation de la performance se basent sur la simulation, les méthodes que nous proposons se basent sur des techniques d'ordonnancement et d'analyse des graphes de précédence. Ces méthodes sont plus rapides et permettent une exploration plus efficace de l'espace des réalisations. Les algorithmes que nous avons développés sont caractérisés par leur faible complexité polynomiale.

L'une des principales tâches de la synthèse de haut-niveau est l'allocation des unités mémoires nécessaires à la mémorisation des résultats intermédiaires produits par les unités de traitement et de contrôle. Nous présentons dans cette thèse une nouvelle approche d'organiser les registres à l'intérieur du chemin de données et de résoudre le problème d'allocation de registres aux variables. Cette approche permet de minimiser le nombre de registres dans le circuit et de réduire sa taille. Pour résoudre le problème d'allocation de registres nous avons utilisé un nouveau paradigme de programmation combinant la programmation à contraintes logiques et l'arithmétique à intervalles.

Cette thèse comprend trois articles publiés et un article soumis à la publication. Les articles publiés sont:

1- I.E. Bennour et E.M. Aboulhamid, "Lower-bounds on the Iteration and Initiation Interval of Functional Pipelining and Loop Folding", paru dans la revue *Design Automation for Embedded Systems*, Vo.1, No. 4, 96

2- I.E. Bennour, M. Langevin et E.M. Aboulhamid,"Performance Analysis for Hardware/Software Cosynthesis", paru dans les actes de la conférence *Canadian Conference on Electrical and Computer Engineering 96*,

3- I.E. Bennour et E.M. Aboulhamid, "Register Allocation using Circular FIFOs" paru dans les actes de la conférence *IEEE International Symposium on Circuits and Systems 96*.

L'article soumis à la publication [BEN96d] est une synthèse bibliographique portant sur le problème d'accélération des boucles posé par la synthèse de haut-niveau des circuits réalisant des traitements itératifs.

• Mots-clés:

Circuits numériques, Graphe de contrôle et de flot de données, Ordonnancement, Estimation de la performance, Programmation en nombres entiers, Programmation logique.

Remerciements

Je témoigne de ma reconnaissance à mon directeur de recherche, monsieur El Mostapha Aboulhamid, professeur au département d'informatique et de recherche opérationnelle à l'université de Montréal. Je le remercie pour sa disponibilité, ses conseils et son encouragement.

Je suis également reconnaissant au Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR) du gouvernement du Québec pour son support financier pendant presque toutes les années de mes études doctorales.

Je remercie tous les membres du laboratoire LASSO pour l'ambiance de travail agréable qu'ils ont su créer.

Je tiens à remercier mon collègue et ami, Dr. Bachir Berkane, pour plusieurs lectures attentives de certains articles de ce manuscrit.

Enfin, tous mes remerciements à ma famille qui n'a cessé de m'encourager.

Table des matières

Liste des figures

Liste des tableaux

Introduction

1. Motivations et thème de la thèse

Le développement technologique de la micro-électronique au cours des vingt dernières années a engendré une utilisation massive des circuits intégrés à très grande échelle (VLSI) dans les applications industrielles. La variété et la complexité croissantes de ces applications ont créé le besoin de développer des outils de CAO aidant le concepteur dès les premières étapes de la synthèse de circuits. Ces outils permettent d'augmenter la productivité et de réduire les erreurs de conception. La figure 1 illustre les principales étapes de synthèse des systèmes numériques:

(1) *La synthèse système* détermine les principales composantes du circuit telles que les interfaces et les unités de traitement, et fixe son architecture globale. Le résultat de cette étape est un ensemble de spécifications comportementales décrivant d'une façon précise la fonctionnalité de chaque composante en faisant abstraction de toute réalisation matérielle.

(2) *La synthèse de haut-niveau* génère une réalisation au niveau transfert de registres (RTL) à partir d'une spécification comportementale. Pour limiter le coût du circuit, des contraintes sur les ressources du circuit (e.g., nombre d'unités fonctionnelles) sont imposées.

(3) *La synthèse logique* transforme une description au niveau RTL en un assemblage d'éléments combinatoires et d'éléments de mémorisation. Les éléments combinatoires sont ensuite optimisés et réalisés à partir d'un ensemble pré-défini de composantes.

(4) *La synthèse au niveau transistor* convertit un réseau de portes logiques en un réseau de transistors et fixe l'emplacement topologique de chaque transistor dans le circuit. Cette étape de synthèse est dépendante de la technologie de fabrication utilisée.

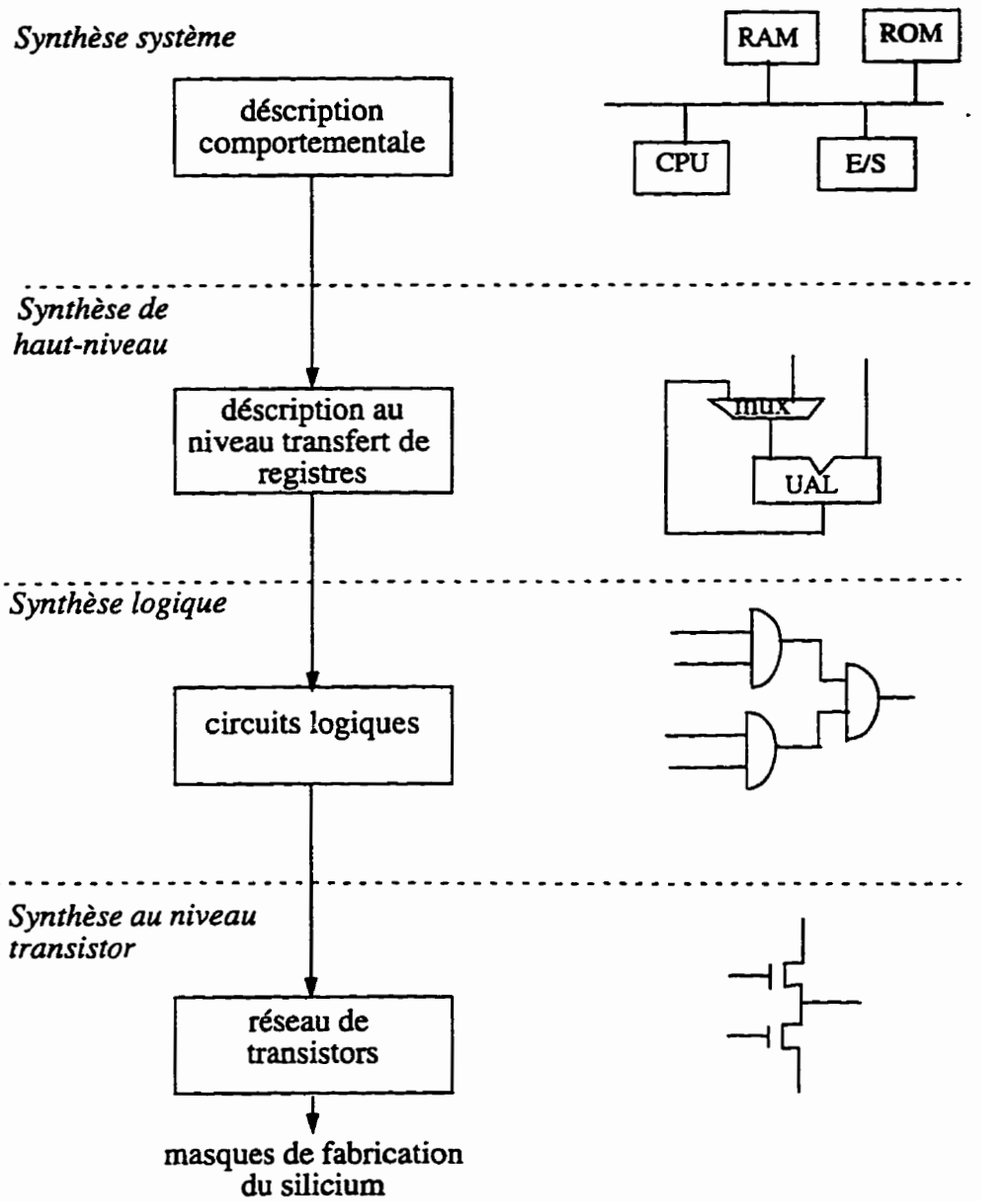


Figure 1. Les différentes étapes de synthèse.

Les étapes de synthèse logique et de synthèse au niveau transistor ont été intégrées dans des nombreux outils de CAO: e.g., Synopsis et Cadence. Par contre pour la synthèse système et la synthèse de haut-niveau (SHN), des efforts restent encore à faire pour intégrer ces étapes dans des outils de CAO industriels.

Les recherches actuelles dans le domaine de la SHN peuvent être reparties en deux groupes. Dans le premier groupe, on s'intéresse à la synthèse de circuits synchrones dont le fonctionnement est contrôlé par une ou plusieurs horloges internes. Les circuits synchrones sont les plus utilisés dans les applications orientées traitement de données. Dans le deuxième groupe, on s'intéresse à la synthèse de circuits asynchrones. Les circuits asynchrones sont surtout utilisés dans les applications orientées contrôle telles que les interfaces.

Les travaux présentés dans cette thèse portent sur la SHN des circuits synchrones. Comme nous l'avons déjà dit, le but de la SHN est de transformer la spécification d'un circuit du niveau comportemental au niveau transfert de registres (RTL) tout en satisfaisant les contraintes de performance et de ressources. Cette transformation passe par deux étapes principales: l'ordonnancement dans le temps des opérations que le circuit doit effectuer, et l'allocation des ressources nécessaires à la réalisation de ces opérations. Dans la littérature, plusieurs heuristiques ont été proposés pour résoudre d'une façon approchée le problème d'ordonnancement qui est NP-complet. Nous n'avons pas l'intention d'en développer d'autres. Par contre, nous allons présenter des algorithmes polynômiaux permettant de calculer des bornes inférieures et supérieures sur la performance qu'un circuit peut atteindre étant données des contraintes de ressources. La borne supérieure correspond à la performance maximale que le circuit peut atteindre indépendamment des heuristiques d'ordonnancement que le concepteur va utiliser. Cette borne est toujours supérieure ou égale à la performance optimum (voir figure 2). À l'aide de la borne supérieure, le concepteur peut connaître à priori si la performance requise peut être atteinte avec les ressources disponibles. La borne inférieure constitue une valeur réalisable de la performance avec les mêmes ressources. Donc, elle donne au concepteur une première estimation de la performance que le circuit peut atteindre.

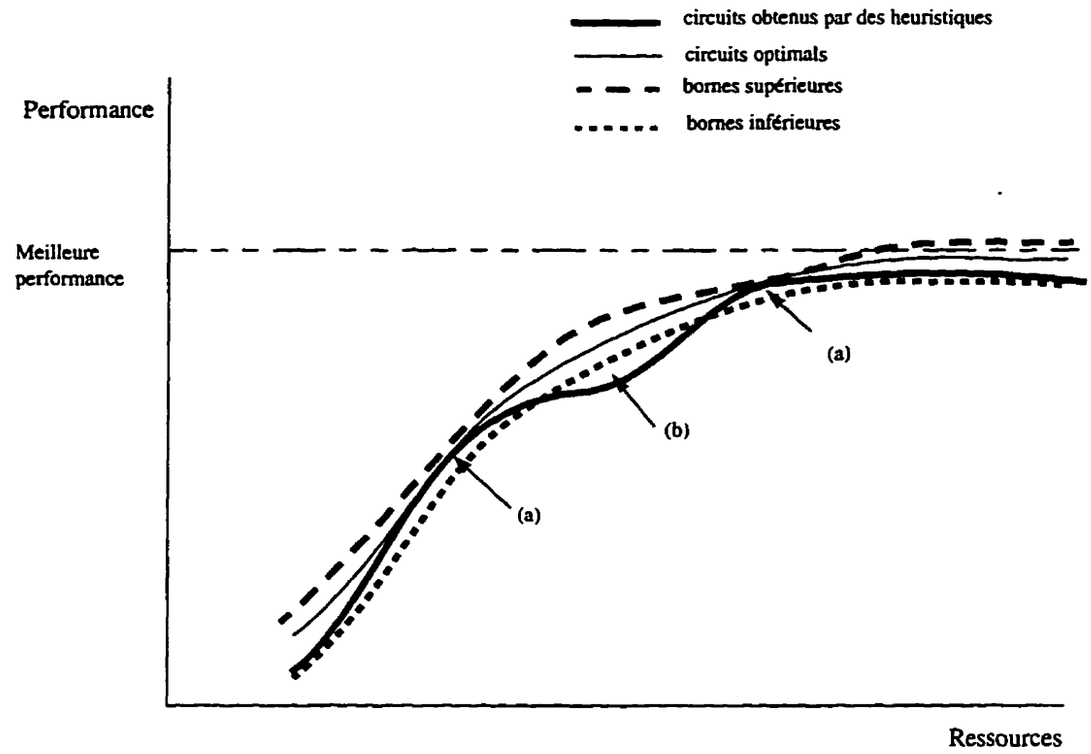


Figure 2. Relation entre les réalisations générées par les différentes méthodes

En plus de l'estimation de la performance d'un circuit avant sa synthèse, les bornes inférieures et supérieures servent à mesurer la qualité des solutions d'ordonnancement produites par les heuristiques. En effet, plus la performance de la solution produite par l'heuristique est proche de la borne supérieure, plus cette solution est proche de l'optimum (cas (a) de la figure 2.). Inversement, plus la performance de la solution heuristique est en dessous de la borne inférieure, plus cette solution est loin de l'optimum (cas (b)).

La seconde étape de la SHN, l'allocation de ressources, est elle même composée de plusieurs sous-étapes dont l'une est l'allocation de registres aux variables intermédiaires produites par les unités de traitement et de contrôle. Les approches traditionnelles d'allocation de registres se limitent à minimiser le nombre de registres dans le circuit sans tenir en compte de l'organisation des registres à l'intérieur du circuit. Or l'organisation des registres a un impact direct sur le nombre de signaux de contrôle et sur la complexité

du chemin de données. Nous présentons dans cette thèse une nouvelle approche d'allocation permettant de minimiser le nombre de registres et d'obtenir des structures mémoires assez régulières et simples à réaliser matériellement.

2. Plan de lecture

Cette thèse est composée de 5 chapitres. Le premier chapitre présente une introduction à la synthèse de haut-niveau et étudie le problème d'accélération des boucles posé par la synthèse des circuits réalisant des traitements itératifs. Un traitement itératif est équivalent à une boucle infinie. Ce type de traitements est fréquent dans les applications de traitement de signaux et de traitement d'images. Le problème d'accélération des boucles est étudié aussi dans les domaines de la recherche opérationnelle et de la compilation où certains résultats peuvent être exploités dans le domaine de la synthèse de circuits. Ce chapitre regroupe les principaux résultats théoriques et pratiques liés à ce problème et expose certaines méthodes pour le résoudre. Ce chapitre donne aussi les motivations des travaux présentés dans les chapitres suivants. Le contenu de ce chapitre est soumis comme un article de synthèse à la revue *Technique et Science Informatique*".

Le deuxième chapitre est constitué d'un article qui paraîtra dans la revue *Design Automation for Embedded Systems*. Cet article présente un ensemble de méthodes algorithmiques qui permettent au concepteur d'évaluer, au cours de l'étape de synthèse de haut-niveau, la performance de différentes réalisations au niveau transfert de registres (RTL) des circuits effectuant des traitements itératifs. Ces méthodes permettent de calculer une borne supérieure sur la performance de toute réalisation d'une spécification comportementale d'un circuit itératif comportant un ensemble de contraintes matérielles portant sur le type et le nombre de ressources disponibles.

Le troisième chapitre est composé d'un article paru dans les actes de la *conférence canadienne en génie électrique et informatique*. Cet article présente des méthodes d'estimation de la performance des réalisations au niveau RTL avec deux extensions par rapport aux méthodes du chapitre précédent: (1) la spécification comportementale du

circuit peut contenir des structures de contrôle complexes telles que des boucles imbriquées et des branchements conditionnels, et des contraintes sur l'architecture du circuit, (2) les méthodes proposées permettent de calculer des bornes inférieures et supérieures sur la performance.

Le quatrième chapitre est composé d'un article paru dans les actes de la conférence *IEEE International Symposium on Circuits and Systems.* Cet article porte sur le problème d'allocation des registres dans la synthèse de haut-niveau de circuits dédiées à des applications spécifiques. Il présente une nouvelle organisation des registres sous forme de files circulaires, adaptée aux applications itératives et qui permet de réduire la taille du circuit.

Nous concluons ce travail (chapitre 5) en mettant l'accent sur l'intérêt des méthodes proposées dans cette thèse et nous discutons certaines directions de recherche.

Chapitre 1.

Les problèmes d'ordonnancement cyclique dans la synthèse de circuits numériques

1.1. Introduction

L'une des principales tâches de la synthèse de haut-niveau est l'ordonnancement dans le temps des opérations que le circuit doit réaliser. Dans ce chapitre nous étudions le problème d'ordonnancement qui apparaît dans la synthèse des circuits réalisant des traitements itératifs. Un traitement itératif (ou répétitif) est équivalent à une boucle infinie. Ce type de traitements est fréquent dans les applications de traitement du signal où un même traitement est répété indéfiniment sur des données différentes. D'une façon informelle, le problème d'ordonnancement cyclique consiste à ordonner dans le temps l'exécution répétitive d'un ensemble d'opérations liées par des contraintes de précédence, en utilisant un nombre limité de ressources. Ce problème est étudié aussi dans les domaines de la recherche opérationnelle et de la compilation où certains résultats peuvent être exploités dans le domaine de la synthèse des circuits numériques.

La synthèse de haut-niveau est présentement un domaine de recherche très actif. Les livres classiques en synthèse de haut-niveau [DEM94, GAJ92, MIC92] n'effleurent les ordonnancements cycliques que superficiellement. Ce chapitre:

- Donne une introduction à la synthèse de haut-niveau.
- Résume le principaux résultats relatifs aux problèmes d'ordonnancement cyclique et qui sont utiles pour la synthèse de circuits.
- Expose certaines méthodes pour résoudre les problèmes d'ordonnancement cyclique.

Ce chapitre est composé de 7 sections. La section 1.2 décrit les principales étapes de la synthèse de haut-niveau. La section 1.3 présente une modélisation graphique des traitements itératifs et les critères d'optimisation dans l'ordonnancement des traitements itératifs. La section 1.4 définit le problème d'ordonnancement cyclique. La section 1.5 présente une synthèse des méthodes de résolution du problème d'ordonnancement cyclique. La section 1.6 traite le problème d'allocation des ressources dans les ordonnancements cycliques. Finalement, la section 1.7 présente certaines directions de recherche.

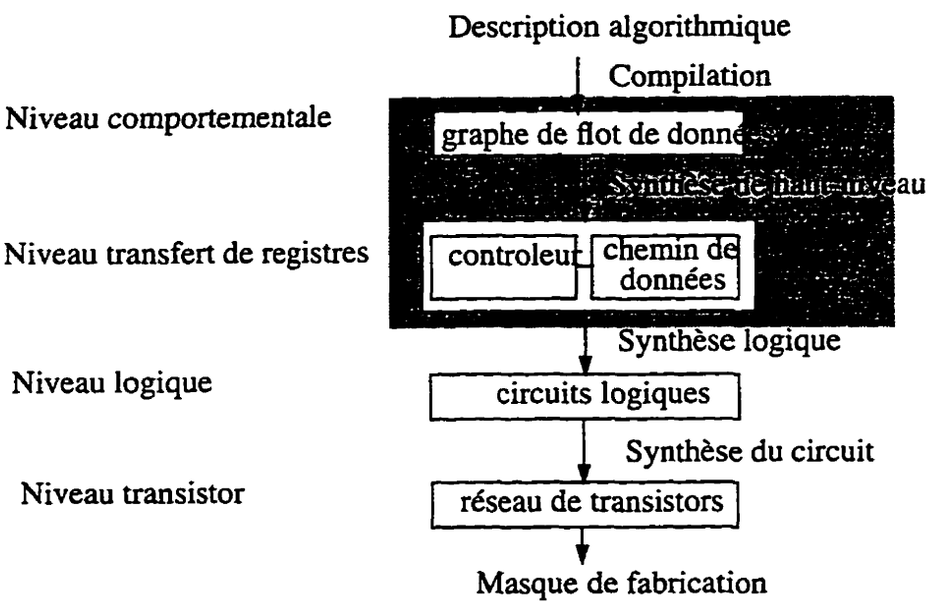


Figure 1.1 Les principales étapes de synthèse.

1.2. La synthèse de haut-niveau

La figure 1.1 montre les principales étapes de conception d'un circuit. Chaque étape transforme la spécification du circuit en introduisant plus de détails. Au plus haut niveau d'abstraction, la spécification décrit la fonctionnalité du circuit en faisant abstraction de toute réalisation matérielle. La spécification est donnée dans un langage de description du matériel tel que VHDL. Elle est ensuite traduite en une représentation graphique appelée graphe de contrôle et de flot de données (GCFD). Dans sa forme la plus simple, le GCFD

est un graphe orienté dont les noeuds représentent les opérations atomiques que le circuit doit réaliser, et les arcs représentent les précédences entre les opérations qui sont dues aux dépendances de données.

Exemple: La figure 1.2.a montre le GCFD correspondant à l'expression

$$z = (2+x) \times (x+y) \times (y+5)$$

La synthèse de haut-niveau est une séquence de tâches qui transforment un GCFD en une description plus détaillée appelée description au niveau transfert de registres. Cette dernière description donne les deux principales composantes formant un circuit, à savoir l'unité de traitement et l'unité contrôle. L'unité de traitement, appelée aussi *chemin de données*, contient les unités fonctionnelles et les registres. L'unité de contrôle détermine à chaque cycle d'horloge quelles opérations du GCFD doivent être exécutées et par quelles unités fonctionnelles.

La synthèse de haut-niveau est composée de quatre tâches dépendantes les unes des autres:

(1) la sélection des unités fonctionnelles,

(2) l'ordonnancement des opérations du GCFD,

(3) l'allocation des registres aux variables, et

(4) l'allocation des bus aux transferts de données.

Dans ce qui suit, nous décrivons brièvement l'objectif de chaque tâche. Pour plus de détails, le lecteur peut se référer à [DEM94, GAJ92].

(1) *Sélection des unités fonctionnelles.* Cette tâche consiste à sélectionner à partir d'une bibliothèque de composantes matérielles les types d'unités fonctionnelles qui seront utilisées pour exécuter les opérations du GCFD. Une unité fonctionnelle est caractérisée principalement par les types d'opérations qu'elle réalise, sa vitesse d'exécution, son coût et sa taille.

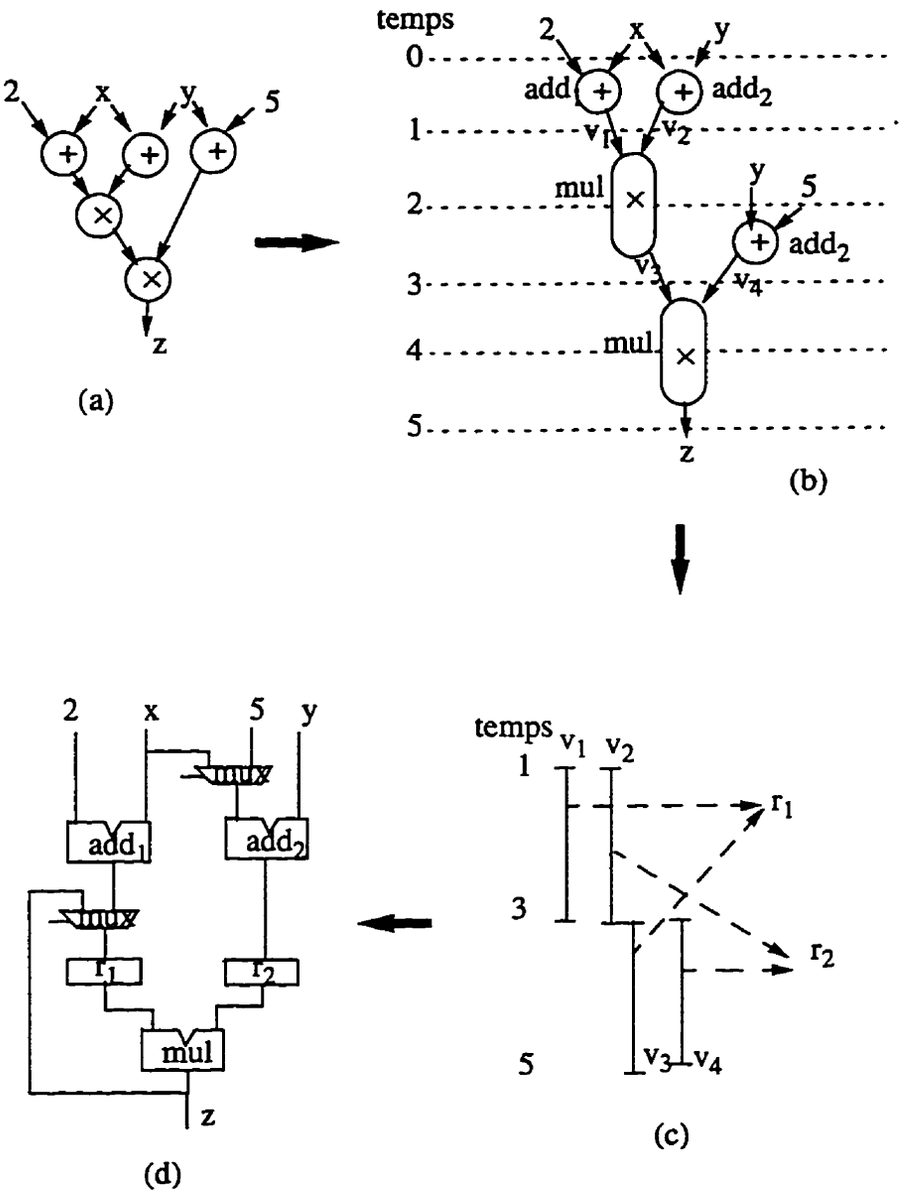


Figure 1.2 Les différentes étapes de la synthèse de haut-niveau: (a) Le GCFD de l'expression $z = (2+x) \times (x+y) \times (y+5)$ *(b) Ordonnancement (c) Les périodes de vie des variables avec une assignation à deux registres (d) Le chemin de données obtenu après la synthèse des connexions.*

(2) *Ordonnancement des opérations du* GCFD. L'objectif de cette tâche est de déterminer un ordre statique d'exécution des opérations du GCFD par les unités fonctionnelles. L'ordre doit respecter les dépendances qui existent entre les opérations. On distingue deux catégories d'ordonnancements: l'ordonnancement sous contraintes de ressources et l'ordonnancement sous contraintes de performance. Dans la première catégorie, le nombre maximal d'unités fonctionnelles à utiliser dans le circuit est fixé d'avance, et l'objectif consiste à minimiser la durée de l'ordonnancement. Dans la deuxième catégorie, la durée de l'ordonnancement est fixée, et l'objectif est de minimiser le nombre d'unités fonctionnelles utilisées.

Exemple: Supposons qu'on dispose de deux additionneurs $\{add_1, add_2\}$ et d'un multiplieur (mul). La durée d'une addition est une unité de temps et celle d'une multiplication est deux unités de temps. La figure 1.2.b montre un ordonnancement réalisable du GCFD.

(3) *Allocation des registres aux variables.* Chaque variable intermédiaire (donnée) produite par une opération du GCFD devrait être sauvegardée dans un registre tant que toutes les opérations utilisant cette donnée n'ont pas été exécutées. La période de vie d'une variable est l'intervalle défini par l'instant de sa création et l'instant de sa dernière utilisation. Les variables qui ont des périodes de vie disjointes peuvent partager le même registre. L'un des principaux critères d'optimisation dans cette tâche est la minimisation du nombre de registres.

Exemple: Le GCFD de la figure 1.2 contient 4 variables intermédiaires $\{v_1, v_2, v_3, v_2\}$. Leurs périodes de vie sont respectivement [1,3], [1,3], [3,5] et [3,5]. Pour sauvegarder ces variables, il faut au moins deux registres. Une solution d'allocation serait d'assigner au premier registre r_1, les variables v_1 et v_3, et au deuxième registre r_2 les variables v_2 et v_4.

(4) *Allocation des bus aux transferts de données.* L'objectif de cette tâche est d'établir les interconnexions entre les unités fonctionnelles et les registres afin de former un chemin de données complet. Le principal critère d'optimisation dans cette tâche est la

minimisation du nombre de connexions et de multiplexeurs.

Exemple: Le chemin de données de l'exemple de la figure 1.2.a obtenu après la synthèse des connexions est montré dans la figure 1.2.d.

1.3. Les traitements itératifs: modélisation et ordonnancement

Nous considérons maintenant le cas où le circuit à synthétiser effectue un traitement itératif. Dans ce type de traitements, deux catégories de dépendances peuvent exister entre les opérations: les dépendances entre les opérations d'une même itération et les dépendances entre des opérations appartenant à des itérations différentes. Cette section présente une modélisation graphique des traitements itératifs et les critères d'optimisation dans l'ordonnancement d'un traitement itératif.

1.3.1. Modélisation des traitements itératifs

Les deux modèles fréquemment utilisés pour représenter les opérations d'un traitement itératif et les contraintes de précédence qui les relient sont les réseaux de Petri temporisés et les graphes de précédence réduits [CAR88, CHR83]. Nous utilisons le deuxième modèle qui est une extension du GCFD.

Un graphe de précédence réduit est un graphe orienté défini par le quadruplet (O, E, H, D) où:

- $O = \{o_i\}$ est l'ensemble de noeuds du graphe; les noeuds représentent les opérations que le circuit doit réaliser. Ces opérations sont dites génériques étant donné qu'elles peuvent être exécutées plusieurs fois sur des données différentes. Une itération correspond à une exécution de l'ensemble des opérations génériques.
- $E \subset O \times O$ est l'ensemble des arcs du graphe reliant toute paire d'opérations ayant une contrainte de précédence entre elles. On note par e_{ij} l'arc de o_i vers o_j.
- $H : E \rightarrow \mathrm{N}$ (l'ensemble des entiers naturels) appelée fonction de hauteur. La valeur $H(e_{ij})$ indique que le résultat de l'opération o_i d'une itération n quelconque est

utilisé par l'opération o_j de l'itération $(n + H(e_{ij}))$. On dit que l'opération o_i est liée à l'opération o_j par une contrainte de précédence intra-itération (resp. par une contrainte de précédence inter-itérations) si $H(e_{ij}) = 0$ (resp. si $H(e_{ij}) > 0$).

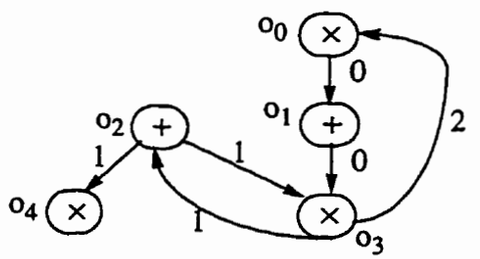
- $D: O \rightarrow N^*$ appelée fonction de durée. La valeur $D(o_i)$ est égale à la durée d'exécution de l'opération o_i.

Dans le reste du chapitre on notera par $\langle o_i, n \rangle$ l'occurrence de l'opération o_i à l'itération n.

Exemple: La figure 1.3 illustre une spécification d'un traitement itératif (a) et le graphe de précédence correspondant (b). Ce graphe de précédence sera utilisé tout le long du chapitre. On supposera que la durée d'une addition est égale à une unité de temps et celle d'une multiplication est égale à deux unités de temps.

De n = 4 à ∞ faire
Debut
o_0: a[n] ← E[n] × d[n-2];
o_1: b[n] ← c_1 + a[n];
o_2: c[n] ← c_2 + d[n-1];
o_3: d[n] ← b[n] × c[n-1];
o_4: S[n] ← c_3 × c[n-1];
Fin

(a)



(b)

Figure 1.3 Un exemple de traitement itératif. (a) Spécification (b) Graphe de précédence

La hauteur et la durée d'un circuit du graphe sont égales, respectivement, à la somme des hauteurs de ses arcs et à la somme des durées de ses sommets. Un graphe de précédence est cohérent s'il ne contient pas de circuit de hauteur nulle. Autrement certaines opérations ne pourront jamais être exécutées sans violer les contraintes de précédence.

Une opération générique est dite *réentrante* (resp. *non réentrante*) si les occurrences

de cette opération peuvent (resp. ne doivent pas) être exécutées en parallèle. Dans le domaine de la synthèse de haut-niveau ainsi que dans le reste du chapitre on suppose par défaut que les opérations sont réentrantes. Notons que la non réentrance d'une opération o_i peut être modélisée dans le graphe de précédence par une boucle e_{ii} de hauteur égale à un.

1.3.2. Ordonnancement des traitements itératifs et critères d'optimisation

Le deux principales mesures de la performance d'un circuit effectuant un traitement itératif sont la fréquence et la latence. La *fréquence* correspond au nombre moyen d'itérations exécutées par unité de temps; plus la fréquence d'un circuit est élevée plus sa performance est élevée. La *latence* est égale à la durée maximale d'exécution d'une itération. Dans les applications où une itération correspond au traitement d'un échantillon en entrée, la latence mesure la rapidité du circuit à traiter un échantillon.

L'objectif de l'ordonnancement d'un traitement itératif est de déterminer un ordre statique des exécutions répétitives des opérations génériques. L'ordre doit respecter les dépendances entre les opérations et les contraintes de ressources s'il y en a. Le premier critère d'optimisation de l'ordonnancement est la maximisation de la fréquence. Le second critère, valable uniquement pour certains types d'applications, est la minimisation de la latence. Pour atteindre des fréquences élevées, l'ordonnancement doit exploiter le deux niveaux de parallélisme présents dans les traitements itératifs: le parallélisme entre les opérations d'une même itération, et le parallélisme entre les opérations qui appartiennent à des itérations différentes.

On distingue, principalement, deux catégories d'ordonnancements cycliques: (1) les ordonnancements périodiques et (2) les ordonnancements K-périodiques. Dans la catégorie (1) toutes les itérations ont le même ordonnancement, et la même durée sépare les débuts des exécutions de deux itérations successives. Dans la catégorie (2) l'ordonnancement de K itérations successives est fixe et se répète à un intervalle régulier.

Exemple: La figure 1.4 montre un ordonnancement périodique de fréquence égale à 1/3 et de latence égale à 5. Un ordonnancement K-périodique du même graphe est donné dans la figure 1.5. La fréquence et la latence de cet ordonnancement sont respectivement de 2/5 et 5.

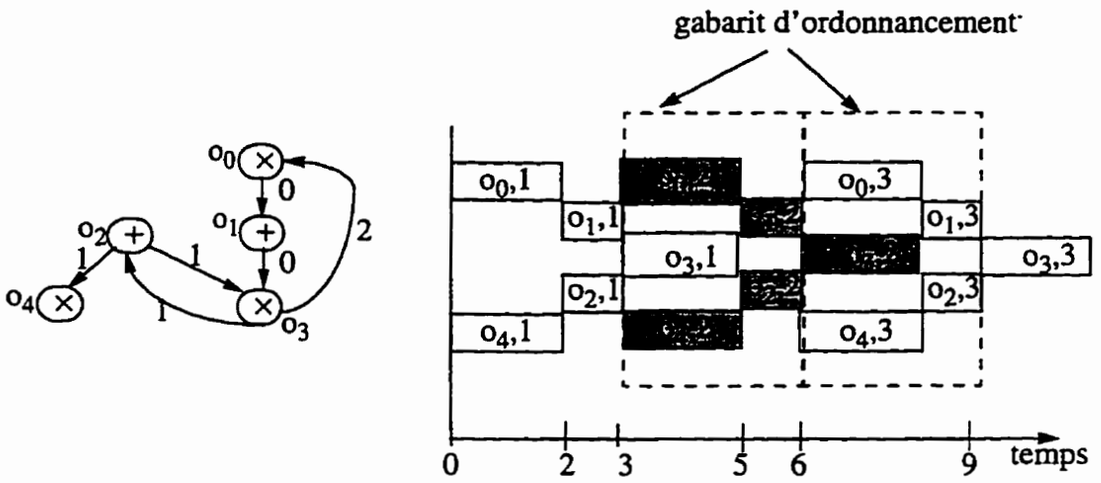


Figure 1.4 Un ordonnancement périodique de fréquence 1/3 et de latence 5.

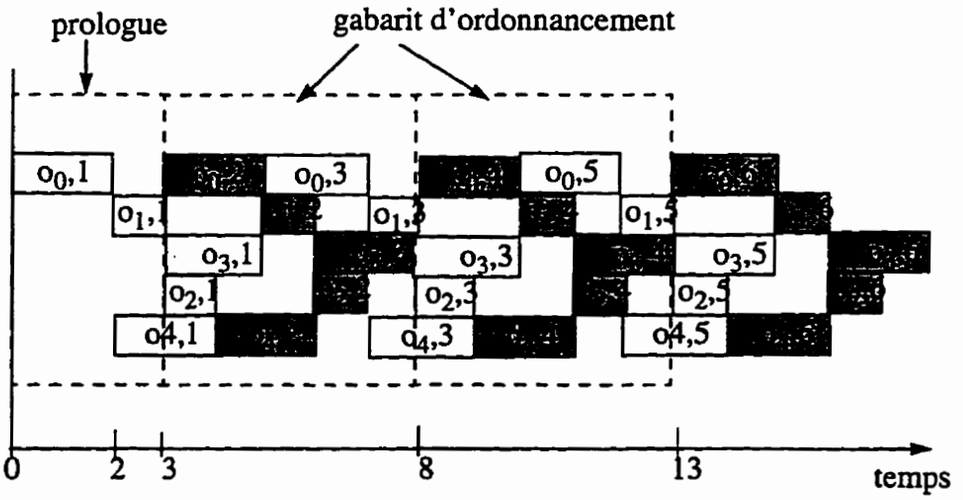


Figure 1.5 Un ordonnancement K-périodique de fréquence 2/5.

L'avantage des ordonnancements périodiques est qu'ils engendrent des architectures matérielles assez régulières. Cependant les ordonnancements K-périodiques permettent d'atteindre des fréquences plus élevées.

Un ordonnancement cyclique est *stable* s'il existe un instant à partir duquel il apparaît un *gabarit d'ordonnancement* qui se répète d'une façon périodique (voir la figure 1.5). Le *prologue* d'un ordonnancement stable est la partie de l'ordonnancement qui précède l'apparition du premier gabarit. Un ordonnancement stable est complètement défini par son prologue et son gabarit. Ainsi, pour réaliser matériellement un ordonnancement cyclique stable il suffit de mémoriser son prologue et son gabarit. La réalisation des ordonnancements qui ne sont pas stables nécessite une mémoire de taille exorbitante et donc ils ne sont pas intéressants.

1.3.3. Le dépliage du graphe de précédence

Le résultat de L dépliages d'un graphe de précédence est un nouveau graphe de précédence représentant les opérations de L itérations successives et les contraintes de précédence intra-itération et inter-itérations qui les relient. Le nombre de noeuds et le nombre d'arcs dans un graphe déplié L fois sont égaux, respectivement, à $L|O|$ et $L|E|$, où $|O|$ et $|E|$ sont, respectivement, le nombre de noeuds et d'arcs dans le graphe initial.

Exemple: La figure 1.6 montre le résultat de deux dépliages.

Notons qu'après un nombre élevé de dépliages, les contraintes de précédence deviennent uniquement entre des opérations d'une même itération ou entre des opérations appartenant au plus à deux itérations successives (autrement dit, les hauteurs des arcs deviennent égales à zero ou un).

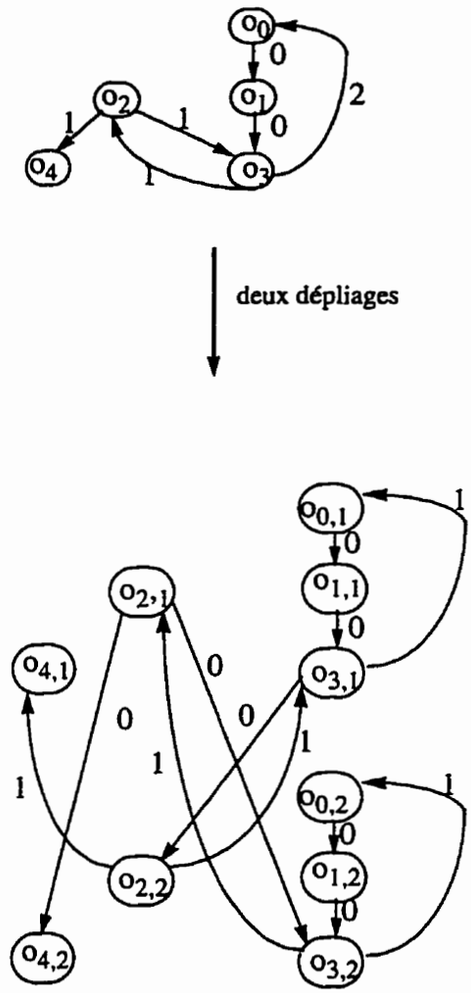


Figure 1.6 Dépliage d'un graphe avec L=2.

L'opération de dépliage a au moins deux utilités pour l'ordonnancement. D'une part, le dépliage rend plus explicite le parallélisme existant entre les opérations appartenant à des itérations différentes. Ceci permet en général d'augmenter l'efficacité (en terme de fréquence) de certains algorithmes d'ordonnancement. D'autre part, certains algorithmes d'ordonnancement sont restreints à des graphes de précédence où les dépendances entre les opérations sont au plus entre deux itérations successives. Le dépliage permet d'utiliser ces algorithmes pour ordonner des graphes quelconques.

Voici un algorithme qui permet de construire le graphe L fois déplié d'un graphe de précédence donné.

1: Pour chaque noeud o_i du graphe initial créer L noeuds $(o_{i,1},\ldots,o_{i,L})$.

2: Pour chaque arc e_{ij} du graphe initial faire:

- Si $H(e_{ij}) = 0$, alors ajouter dans le nouveau graphe des arcs $o_{i,q} \rightarrow o_{j,q}$ (de $q = 1$ à L) avec une hauteur nulle.

-Si $H(e_{ij}) < L$, alors ajouter des arcs $o_{i,q-H(e_{ij})} \rightarrow o_{j,q}$ (de $q = H(e_{ij}) + 1$ à L) avec une hauteur nulle. Et ajouter des arcs $o_{i,L-H(a_{ij})+q} \rightarrow o_{j,q}$ (de $q = 1$ à $H(e_{ij})$) avec une hauteur égale à un.

-Si $H(e_{ij}) \geq L$, ajouter des arcs $o_{i,\left\lceil \frac{H(e_{ij})-q+1}{L} \right\rceil L-H(e_{ij})+q} \rightarrow o_{j,q}$ avec une hauteur égale à $\left\lceil \frac{H(e_{ij})-q+1}{L} \right\rceil$ (de $q = 1$ à L) .

1.4. Spécification du problème d'ordonnancement cyclique et calcul de la fréquence maximale

Dans la première partie de cette section nous donnons les définitions précises du problème d'ordonnancement cyclique et des ordonnancements K-périodiques et périodiques. Dans la seconde partie, nous étudions les facteurs qui limitent la fréquence d'un ordonnancement.

1.4.1. Modèle de ressources et définitions

Le modèle de ressources. On considère que chaque opération générique est affectée au préalable à un type d'unité fonctionnelle. Plusieurs opérations peuvent être affectées au même type d'unité fonctionnelle. Il peut y avoir plusieurs instances d'un même type

d'unité fonctionnelle. Une unité fonctionnelle est dite *homogène* si elle peut exécuter n'importe quelle opération générique.

L'ordonnancement d'un graphe de précédence. Soit $G = (O, E, H, D)$ un graphe de précédence et soit Q^+ l'ensemble des rationnels positifs. Un ordonnancement cyclique de G est une fonction $s: O \times \mathrm{N}^* \rightarrow Q^+$ qui associe à chaque occurrence $\langle o_i, n \rangle$ l'instant de son début exécution. Un ordonnancement s est réalisable ssi les deux contraintes suivantes sont vérifiées:

(1) Contraintes de précédence:

$$\forall e_{ij} \in E, \forall n \in \mathrm{N}^* \bullet \; s\langle o_j, H(e_{ij}) + n \rangle \geq s\langle o_i, n \rangle + D(o_i) \qquad (1)$$

(2) Contraintes de ressources: il existe une allocation des unités fonctionnelles aux opérations sans qu'il y ait un conflit de ressources.

La fréquence d'un ordonnancement cyclique s est égale à

$$\lim_{n \to \infty} \frac{n}{\max\limits_{o_i} \{ s\langle o_i, n \rangle + D(o_i) \}}$$

Le problème d'ordonnancement cyclique (POC). Etant donné un graphe de précédence, une affectation des opérations génériques aux types d'unité fonctionnelles et un nombre d'instances de chaque type d'unité, le POC est de déterminer un ordonnancement réalisable de fréquence maximale.

Ordonnancement K-périodique. Un ordonnancement s est dit K-périodique s'il existe deux entiers n_0, K et un rationnel positif P tel que:

$$\forall n \geq n_o, \forall o_i \in O \bullet s\langle o_i, n + K \rangle = s\langle o_i, n \rangle + P$$

K et P sont appelés, respectivement, la périodicité et la période de s. La fréquence de s est égale à $\frac{K}{P}$.

Ordonnancement périodique. Un ordonnancement s est périodique s'il existe un rationnel positif P tel que

$$\forall n \geq 2, \forall o_i \in O \bullet s\langle o_i, n\rangle = s\langle o_i, n-1\rangle + P \tag{2}$$

La fréquence de s est égale à $\frac{1}{P}$. Un ordonnancement périodique est un ordonnancement K-périodique de périodicité égale à un.

1.4.2. Facteurs affectant la fréquence

Deux facteurs limitent la fréquence maximale d'un traitement itératif: les contraintes de ressources et les dépendances entre les opérations. La valeur de la fréquence maximale due aux contraintes de ressources, notée F_{ress}, est donnée par l'expression suivante:

$$F_{ress} = \min_{h} \left\{ \frac{m_h}{\sum_{o_i \in O_h} D(o_i)} \right\} \tag{3}$$

où m_h est le nombre d'unités fonctionnelles de type h disponibles, O_h est l'ensemble d'opérations génériques qui doivent être exécutées sur des unités de type h, et $D(o_i)$ est la durée de l'opération générique o_i.

La fréquence maximale due aux dépendances entre les opérations est appelée *fréquence critique* et notée F_{crit}. La valeur de F_{crit} est donnée par la formule suivante [CHR83]:

$$F_{crit} = \min_{C_k} \left\{ \frac{\sum_{e_{ij} \in C_k} H(e_{ij})}{\sum_{o_i \in C_k} D(o_i)} \right\} \tag{4}$$

où C_k est un circuit élémentaire dans le graphe précédence. Cette formule découle de l'équation (1). Si le graphe précédence ne contient pas de circuits, alors théoriquement la fréquence critique est infinie et toutes les itérations peuvent s'exécuter en parallèle s'il y

a suffisamment de ressources. Un circuit est dit critique si le rapport entre sa hauteur et sa durée est égale à la fréquence critique.

Exemple: Le graphe de précédence de la figure 1.3 contient deux circuits $C_1 = o_0 \rightarrow o_1 \rightarrow o_3 \rightarrow o_0$, $C_2 = o_2 \rightarrow o_3 \rightarrow o_2$

Si on suppose que la durée d'une addition est égale à une unité de temps et celle d'une multiplication est égale à deux unités de temps, alors la valeur de la fréquence critique serait égale à $\min\left(\frac{2}{5}, \frac{2}{3}\right) = \frac{2}{5}$. Le circuit C_1 est critique.

La fréquence maximale réalisable, notée F_{max}, est bornée par le minimum de F_{crit} et F_{ress}:

$$F_{max} \leq \min (F_{ress}, F_{crit}) \tag{5}$$

1.4.3. Méthode de calcul de la fréquence critique

La fréquence critique d'un graphe de précédence peut être calculée en un temps polynômial. Cette section présente une méthode simple et efficace pour calculer la fréquence critique; d'autres méthodes sont décrites dans [ZAK89, GER92]. Notons que le calcul de la fréquence critique directement à partir de la formule (4) peut prendre théoriquement un temps exponentiel vu que le nombre de circuits élémentaires dans un graphe peut être exponentiel.

Bornes inférieure et supérieure de la fréquence critique: À partir de la formule (4), on peut déduire que

$$F_{crit} \in \left[\frac{1}{|O| \cdot d_{max}}, h_{max}\right]$$

$$\text{où, } d_{max} = \max_{o_i} \{D(o_i)\} \cdot h_{max} = \max_{e_{ij}} \{H(e_{ij})\}$$

La borne inférieure de F_{crit} correspond au cas où le graphe contient un circuit qui passe par tous les noeuds et que tous les arcs du circuit ont une durée égale à d_{max} et une

hauteur nulle, à l'exception d'un arc qui a une hauteur égale à un. La borne supérieure correspond au cas où le graphe contient un circuit qui passe par tous les noeuds et que tous les arcs du circuit ont une hauteur égale à h_{max} et une durée égale à un.

La méthode de calcul de la fréquence critique est basée sur la propriété [DON92] suivante qui découle de la contrainte de l'équation (1).

Soient $G = (O, E, H, D)$ *un graphe de précédence et* F *un rationnel positif.* F *est une fréquence réalisable de* G *si le graphe* $\hat{G} = (O, E, \hat{H}, D)$ *où*

$$\hat{H}(e_{ij}) = D(o_i) - \frac{1}{F} \cdot H(e_{ij})$$

ne contient pas de circuit de hauteur positive.

Ainsi, le calcul de la fréquence critique revient à la recherche de la plus grande valeur dans l' intervalle $[\frac{1}{|O| \cdot d_{max}}, h_{max}]$ qui vérifie la propriété précédente. La complexité de cette méthode en temps est de $O(|O||E| \log(|O| d_{max}))$ [DON92].

1.5. Le problème d'ordonnancement cyclique (POC): complexité et méthodes de résolution

La difficulté dans la résolution du POC est due principalement aux contraintes de ressources et à la présence de circuits dans le graphe de précédence. Dans cette section nous étudions le POC en distinguant les trois cas suivants: (1) cas où il n'y a pas des contraintes de ressources, (2) cas où le graphe de précédence est acyclique, (3) cas général où il y a des contraintes de ressources et des circuits dans le graphe de précédence. Les cas particuliers (1) et (2) ont été distingués car ils sont fréquents dans la synthèse des circuits numériques. En fait, pour certaines applications qui requièrent des fréquences élevées les contraintes sur les ressources sont secondaires, et plusieurs traitements itératifs ne contiennent pas des contraintes de précédence inter-itérations, et donc ils ont des graphes de précédence acycliques.

1.5.1. Le POC sans contraintes de ressources

Dans le cas où il n'y a pas de contraintes de ressources, le POC est calculable en un temps polynômial. La fréquence optimale est égale à la fréquence critique. D'autre part, il existe toujours des ordonnancements K-périodiques à valeur entière et de fréquence optimale. En plus, il existe toujours des ordonnancements périodiques, pas nécessairement à valeur entière, de fréquence optimale. On dit que les ordonnancements périodiques sont *dominants* dans le cas où il n'y a pas de contraintes de ressources. Dans ce qui suit, nous décrivons une méthode simple pour calculer ces deux types d'ordonnancements.

D'après l'équation (2), les ordonnancements périodiques de fréquence maximale sont de la forme:

$$\forall n \geq 2, \forall o_i \in O.\ s\langle o_i, n\rangle = s\langle o_i, n-1\rangle + \frac{1}{F_{crit}} \tag{6}$$

Une fois que la valeur de F_{crit} est calculée, les valeurs $s\langle o_i, 1\rangle$ peuvent être déduites facilement. Ces valeurs doivent satisfaire la contrainte de l'équation (1). En combinant les équations (1) et (6), on obtient:

$$\forall e_{ij} \in E.\ \ s\langle o_i, 1\rangle - s\langle o_j, 1\rangle \leq \frac{H(e_{ij})}{F_{crit}} - D(o_i)$$

Ce système est une version du problème du plus court chemin. Il peut être résolu par l'algorithme de Bellman-ford [COR90] en un temps $O(|O||E|)$.

Une fois l'ordonnancement périodique calculé, il suffit d'appliquer le théorème suivant [HAN94] pour obtenir un ordonnancement K-périodique de fréquence maximale. Ce théorème n'est valable qu'en l'absence de contrainte de ressources.

Si s est un ordonnancement périodique, alors s^* défini comme suit:

$$\forall o_i \in O, \forall n \geq 1.\ s^*\langle o_i, n\rangle = \lfloor s\langle o_i, n\rangle \rfloor$$

est un ordonnancement K-périodique et de même fréquence que s.

Dans le cas où il n'y a pas des contraintes de ressources, d'autres méthodes polynomiales ont été proposées dans [SCH89, ZAK89, IWA90, CHA93a, JEN94] pour calculer des ordonnancements K-périodiques, à valeur discrète et de fréquence maximale. La méthode de Iwano *et al.* [IWA90] est efficace car elle permet d'obtenir des ordonnancements de fréquence maximale et de gabarit de taille minimale.

L'ordonnancement cyclique au plus tôt. Considérons le mode d'ordonnancement qui consiste à ordonner les instances des opérations génériques au plus tôt: l'exécution d'une opération est activée dès que l'exécution de ses prédécesseurs est terminée. Un tel ordonnancement est dit *au plus tôt*. Carlier *et al.* [CHR88] ont montré que dans le cas où les opérations sont non réentrantes et que le graphe de précédence est fortement connexe, les ordonnancements *au plus tôt* ont une structure K-périodiques; avec une périodicité K égale au produit des hauteurs des circuits critiques et une période égale à $K \cdot F_{crit}$. Ce résultat est aussi valable pour certains graphes faiblement connexes [CHR88]. Les ordonnancements *au plus tôt* ont l'avantage d'avoir une latence minimale en plus de l'exécution au plus tôt des itérations. Par contre la taille du gabarit de ces ordonnancements peut être trop grande.

L'ordonnancement cyclique avec date limite. On suppose maintenant que la date de la fin d'ordonnancement de chaque itération n est soumise à une date limite Δ_n. Il est intéressant de savoir le moment le plus tard $t_\Delta\langle o_i, n\rangle$ de l'ordonnancement de l'opération $\langle o_i, n\rangle$ tel que les dates limites soient respectées. Ce problème a été étudié dans [CHR91]. Il a été montré que dans le cas particulier où chaque Δ_n est égale au moment le *plus tôt* de la fin d'exécution de l'itération n, l'ordonnancement $t_\Delta = \{t_\Delta\langle o_i, n\rangle\}$ a une structure K-périodique avec une périodicité K égale au produit des hauteurs des circuits critiques et une période égale à $K \cdot F_{crit}$.

1.5.2. Le POC avec des graphes de précédence acycliques

Munier a montré dans [MUN91] que si le graphe de précédence ne contient pas de

circuits alors le POC est polynômial. Notons que le problème de la minimisation de la latence pour une fréquence fixée est NP-complet même si le graphe de précédence est acyclique [GRO92].

1.5.3. Le POC avec contraintes de ressources et graphe de précédence cyclique

Dans son cas général le POC est un problème NP-complet, même si le graphe de précédence est composé uniquement d'un circuit [HAN94]. Un algorithme polynômial proche de l'optimum pour résoudre le problème à m unités fonctionnelles homogènes est donné dans [GAS92]. La fréquence de l'ordonnancement obtenue par cet algorithme est égale dans le pire cas au deux tiers de la fréquence optimale.

Plusieurs méthodes heuristiques ont été proposées dans la littérature pour résoudre le POC. Nous décrivons dans ce qui suit les trois approches d'ordonnancement fréquemment utilisées.

1*ère approche*. Elle consiste à fixer un même ordonnancement pour toutes les itérations. Cet ordonnancement doit satisfaire les contraintes de précédence intra-itération et les contraintes de ressources. Puis les moments de début d'exécution des itérations sont calculés de sorte que les contraintes de précédence inter-itérations et les contraintes de ressources soient respectées. Cette approche est souvent utilisée avec le modèle des tables de réservation [KOG81]. Les ordonnancements obtenus ont généralement une structure K-périodique. Des algorithmes qui utilisent cette approche sont décrits dans [SU87, LIU89, HAN90].

2*ème approche*. Cette approche est itérative et consiste à fixer la fréquence à une certaine valeur cible inférieure ou égale à la borne supérieure donnée par l'équation (5), puis à chercher un ordonnancement périodique ayant cette fréquence. Si au bout d'un certain temps la recherche échoue, la fréquence cible est réduite et le processus est réitéré. Théoriquement, cette approche permet d'atteindre la fréquence maximale si le graphe de précédence est suffisamment déplié. Des algorithmes qui utilisent cette approche sont

décrits dans [GRO92, WAN93, LEE94].

3ème approche. Il s'agit d'une approche constructive. On commence avec une solution d'ordonnancement réalisable puis on la modifie dans le but d'améliorer la fréquence. Chao *et al.* ont montré dans [CHA93b] que la technique de resynchronisation pouvait être utilisée pour déterminer des ordonnancements efficaces d'une manière constructive. Cette technique est détaillée dans la section suivante.

1.5.4. La resynchronisation et son utilisation dans l'ordonnancement

1.5.4.1. Définition de la resynchronisation

La technique de resynchronisation a été développée initialement pour réduire la période d'horloge dans les circuits synchrones [LEI91]. Elle consiste à réorganiser le graphe de précédence en modifiant les hauteurs des arcs.

Soit $G = (O, E, H, D)$ un graphe de précédence et soit φ une fonction de O dans Z (l'ensemble des entiers) dite fonction de resynchronisation. On appelle graphe resynchronisé de G par rapport à φ le graphe $G_\varphi = (O, E, H_\varphi, D)$ où

$$\forall e_{ij} \in E \;.\; H_\varphi(e_{ij}) = H(e_{ij}) + \varphi(o_j) - \varphi(o_i)$$

Exemple: La figure 1.7 montre le graphe obtenu après la resynchronisation: $\varphi(o_0) = \varphi(o_1) = \varphi(o_4) = -2 ; \varphi(o_2) = \varphi(o_3) = -1$.

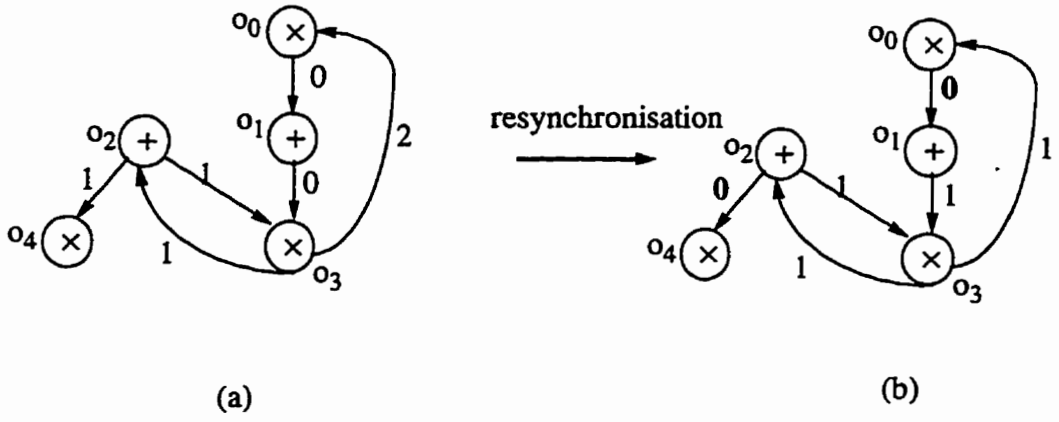


Figure 1.7 Graphe de précédence après une resynchronisation.

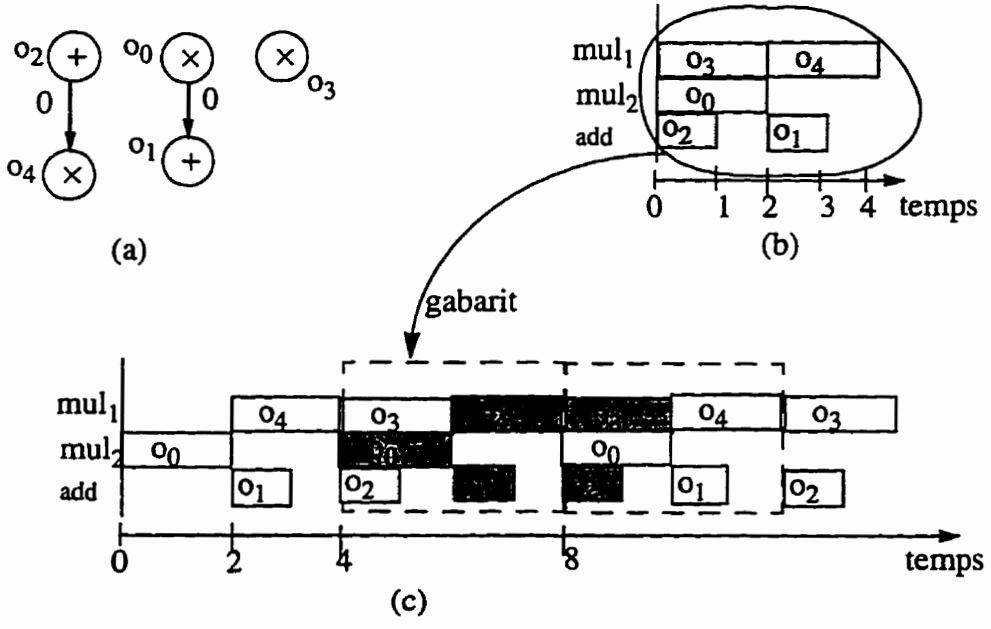


Figure 1.8 (a) Le graphe resynchronisé de la figure 1.7.b avec seulement les dépendances intra-itération (b) Un ordonnancement non cyclique de ce graphe (c) L'ordonnancement cyclique dérivé à partir de l'ordonnancement non cyclique

1.5.4.2. Relation entre resynchronisation et ordonnancement

Soit G un graphe de précédence et soit G_φ le graphe obtenu par une resynchronisation de G. On peut démontrer [CHA93b] que tout ordonnancement des opérations de G_φ qui satisfait uniquement les contraintes de précédence intra-itération (arcs de hauteurs nulles) et les contraintes de ressources est un gabarit d'un ordonnancement périodique de G. Cette propriété montre que la solution d'un problème d'ordonnancement cyclique peut être obtenue en résolvant un problème d'ordonnancement simple (non cyclique) précédé d'une resynchronisation. La difficulté réside dans le choix de la resynchronisation à appliquer.

Exemple: Le graphe de la figure 1.8.a est le graphe resynchronisé de la figure 1.7.b avec seulement les dépendances intra-itération. Supposons qu'on dispose de deux multiplicateurs et d'un additionneur. Un ordonnancement qui satisfait les dépendances intra-itération et les contraintes de ressources est donné dans la figure 1.8.b. Cet ordonnancement est le gabarit de l'ordonnancement cyclique périodique de la figure 1.8.c.

1.5.4.3. Une heuristique d'ordonnancement utilisant la resynchronisation

Dans ce qui suit on présente une heuristique illustrant l'utilisation de la resynchronisation dans un algorithme d'ordonnancement. L'heuristique consiste à effectuer d'une façon répétitive la resynchronisation suivante

$$\forall o_i \in O.\ \varphi(o_i) = \left\{ \begin{array}{l} -1 \text{ si } \forall e_{ji}, H(e_{ji}) > 0 \\ 0, \text{autrement} \end{array} \right\}$$

suivie d'un ordonnancement, jusqu'à l'obtention d'une fréquence satisfaisante. L'heuristique est composée de trois étapes:

(a) Appliquer sur le graphe de précédence courant G la resynchronisation φ. Le graphe résultant est G_φ.

(b) Trouver un ordonnancement des opérations de G_φ qui satisfait uniquement les contraintes de précédence intra-itération et les contraintes de ressources. Le critère d'optimisation est la latence de l'ordonnancent. À ce niveau, on peut utiliser

les algorithmes d'ordonnancement existants pour résoudre les problèmes d'ordonnancement non cycliques.

Rappelons que l'ordonnancement obtenu est le gabarit d'un ordonnancement cyclique de G.

(c) Si la latence obtenue à l'étape (b) n'est pas satisfaisante, reprendre l'étape (a) avec $G \leftarrow G_{\varphi}$.

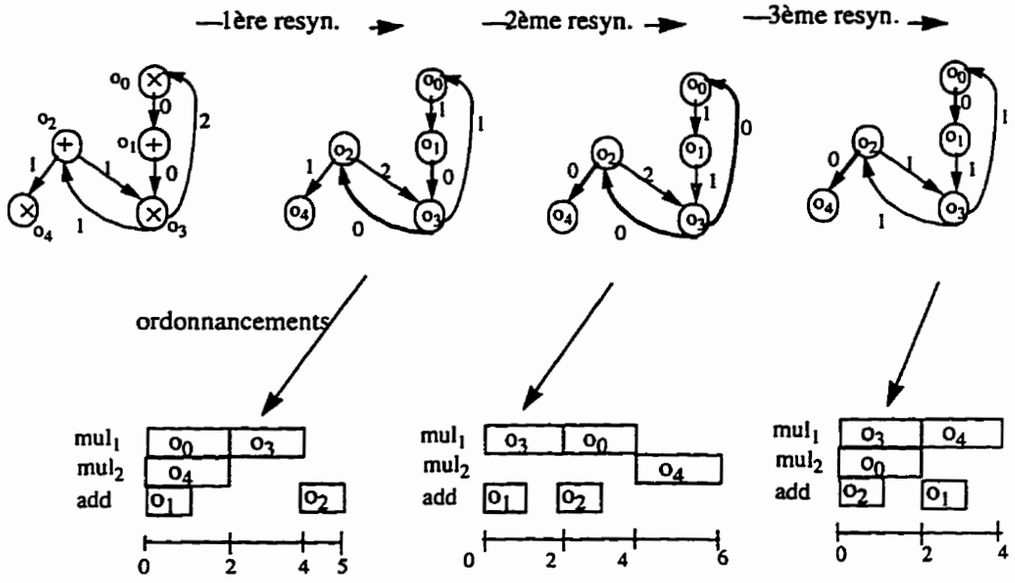


Figure 1.9 *Illustration de l'heuristique d'ordonnancement (nous avons omis* les durées des arcs).
1ère resyn. $\varphi(o_0) = \varphi(o_2) = \varphi(o_4) = -1 ; \varphi(o_1) = \varphi(o_3) = 0 ;$
2ème resyn. $\varphi(o_0) = \varphi(o_1) = \varphi(o_4) = -1 ; \varphi(o_2) = \varphi(o_3) = 0 ;$
3ème resyn. $\varphi(o_1) = \varphi(o_3) = -1 ; \varphi(o_0) = \varphi(o_2) = \varphi(o_4) = 0 ;$

Illustration de l'heuristique. La figure 1.9 montre les trois itérations (une resynchronisation suivie d'un ordonnancement) effectuées avant l'obtention du gabarit de taille minimale. L'ordonnancement cyclique de fréquence maximale dérivé à partir du gabarit de taille minimale est celui de la figure 1.8.c.

1.6. Le problème d'allocation de ressources dans les ordonnancements cycliques

Dans la synthèse de haut-niveau, les deux tâches qui viennent après l'ordonnancement sont l'allocation des unités fonctionnelles aux opérations et l'allocation des registres aux variables. Ces deux tâches sont discutées dans cette section.

1.6.1. Allocation des unités fonctionnelles

L'allocation des unités fonctionnelles consiste à assigner à chaque opération l'unité fonctionnelle qui doit l'exécuter. Pour un même ordonnancement, il peut exister plusieurs solutions d'allocation qui utilisent le même nombre d'unités fonctionnelles. Nous présentons dans ce qui suit certaines catégories d'allocation simples à réaliser matériellement.

Notons par $U = \{U_0, U_1, \ldots, U_{m-1}\}$ l'ensemble des unités fonctionnelles disponibles, et par $r: O \times \mathrm{N}^* \rightarrow U$ la fonction d'allocation qui assigne à chaque opération $\langle o_i, n\rangle$ une unité fonctionnelle.

Une allocation r est de type permutation, s'il existe une permutation $p: U \rightarrow U$ telle que:

$$\forall o_i \in O,\ \forall n > 1 \,.\, r\langle o_i, n\rangle = p\,(r\langle o_i, n-1\rangle)$$

L'allocation r est complètement définie par la permutation p et les valeurs initiales de $r\langle o_i, 1\rangle, \forall o_i$. Par conséquent, pour réaliser matériellement r il suffit de mémoriser la permutation p et l'ensemble $\{r\langle o_i, 1\rangle\}$.

Exemple: La figure 1.10 montre un ordonnancement périodique qui utilise trois ressources et une allocation de type permutation. L'allocation est définie par:

$p\,(U_0) = U_1\,, p\,(U_1) = U_0\,, p\,(U_2) = U_2$

$r\langle o_0, 1\rangle = r\langle o_3, 1\rangle = U_1\,, r\langle o_1, 1\rangle = r\langle o_2, 1\rangle = U_2\,, r\langle o_4, 1\rangle = U_0$

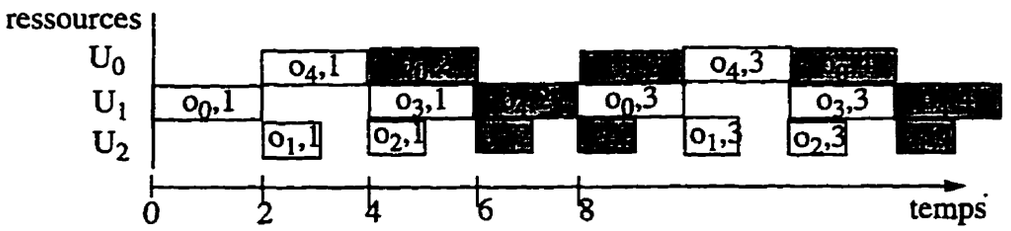


Figure 1.10 Un ordonnancement périodique avec une allocation de type permutation.

Voici trois permutations particulières:

- *La permutation identité*:

$$\forall U_i \in U . p(U_i) = U_i$$

Les allocations définies par la permutation identité sont simples à réaliser car la même unité fonctionnelle est allouée à toutes les instances d'une opération générique. Par contre, elles limitent la fréquence d'exécution des itérations.

-*La permutation circulaire*:

$$\forall U_i \in U . p(U_i) = U_{(i+1) \bmod m}$$

où m est le nombre d'unités fonctionnelles. Hanen *et al.* [HAN95] ont montré que les allocations définies par une permutation circulaire sont dominantes dans les ordonnancements périodiques à opérations non réentrantes. En d'autres termes il existe toujours un ordonnancement périodique de fréquence maximale avec une allocation de ressources circulaire. Notons que dans ce type d'allocation on suppose que toutes les unités fonctionnelles sont homogènes.

- *La permutation cyclo-statique* [SCH85]:

$$\forall U_i \in U . p(U_i) = U_{(i+d) \bmod m}$$

où d est une constante positive, appelée facteur de déplacement des ressources. La permutation circulaire est un cas particulier de la permutation cyclo-statique.

Comme direction de recherche, il serait intéressant d'étudier le problème d'ordonnancement cyclique sous des contraintes relatives au mode d'allocation des unités fonctionnelles aux opérations en plus des contraintes sur le nombre d'unités disponibles.

1.6.2. Allocation des registres aux variables

Exemple: Le graphe de la figure 1.11.a contient 4 variables intermédiaires. Les durées de vie de ces variables obtenues après l'ordonnancement des opérations sont représentées par des segments dans la figure 1.11.c. À titre d'exemple, la variable v_2 est modifiée par l'opération $<o_2,1>$ à l'instant 5, et elle est utilisée une itération plus tard par l'opération $<o_3,2>$ dont l'exécution se termine à l'instant 10. Remarquons qu'il y a toujours deux occurrences de la variable v_2 qui sont en vie en même temps, ce qui nécessite au moins deux registres pour les sauvegarder. Le nombre total de registres nécessaires est 4.

Etant donnés un graphe de précédence et son ordonnancement, le problème d'allocation des registres consiste à trouver une assignation des variables aux registres qui minimise le nombre total de registres. Ce problème est NP-complet [DEM94]; il est polynômial dans les ordonnancements non-cycliques. Des heuristiques d'allocation sont données dans [HEN92, RAU92, ALO94].

Le nombre de registres peut être aussi donné comme une contrainte et le problème serait de trouver un ordonnancement de fréquence maximale sous cette contrainte. Peu de travaux [HAN90] ont traité ce problème.

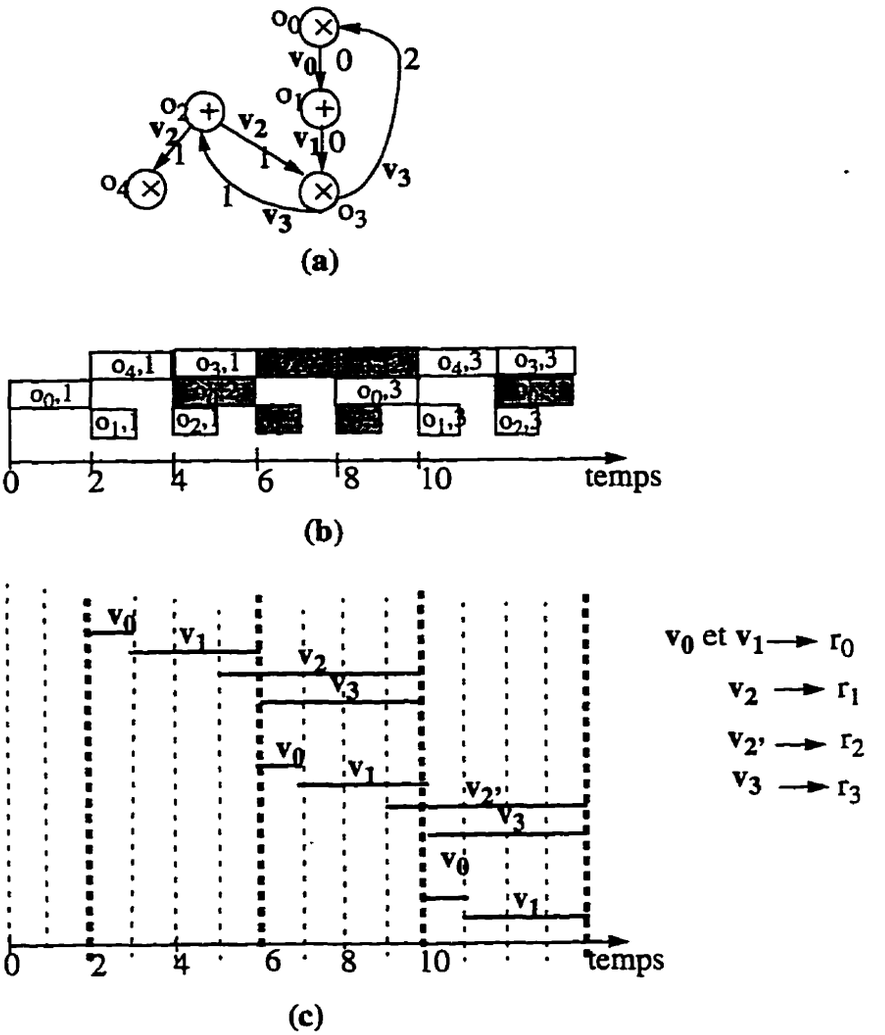


Figure 1.11 Un exemple d'allocation des registres aux variables. (a) Les variables du graphe de précédence (b) Un ordonnancement (c) Durée de vie des variables et allocation des registres.

Table 1.1: Complexité des problèmes liés aux ordonnancements cycliques

Problèmes	Contraintes de ressources	Circuits dans le graphe de précédence	Complexité	Référ. (algorithmes exacts ou heuristiques)
Calcul de la fréquence critique	X†	1	Polynomiale	[KAR78, ZAK89, GER92]
Maximisation de la fréquence (POC)	1	1	NP-complet	[SU87, LAM88, LIU89, GOO90, HAN90, AIK91, GAS92, CHA93b]
Maximisation de la fréquence	1	0	Polynomiale	[MUN91]
Maximisation de la fréquence	0	1	Polynomiale	[SCH89, ZAK89, IWA90, PAR91, DON92, CHA93a, JEN94]
Minimisation de la latence pour une fréquence donnée	1	X	NP-complet	[PAR88, GRO92, HWA91, WAN93, LEE94]
Minimisation des registres pour un ordonnancement donné	X	1	NP-complet	[HEN92, RAU92, ALO94]

† 1= contrainte présente, 0 = contrainte absente, X= 0 ou 1.

1.7. Conclusions et directions de recherche

Dans cette étude nous avons présenté les résultats que nous avons considérés pertinents pour les problèmes de l'ordonnancement cyclique dans le cadre de la synthèse de haut-niveau. Le tableau 1.1 résume la complexité de ces problèmes et donne les références aux méthodes existantes pour les résoudre. Ces résultats ont été empruntés aux trois domaines où ce problème surgit, à savoir la compilation, la recherche opérationnelle (RO) et la synthèse de haut-niveau (SHN). Ce dernier domaine possède ses

caractéristiques propres qui le différencient des deux autres. Alors que la compilation suppose des processeurs séquentiels ou à la rigueur des multiprocesseurs homogènes, la SHN va s'intéresser à une granularité plus fine du parallélisme. Le domaine de la RO foisonne de résultats intéressants mais certaines hypothèses telles que la non-réentrance limitent l'utilisation immédiate de ces résultats, il serait profitable de réadapter ces résultats au domaine de la SHN. D'autre part, les boucles ne sont qu'un cas particulier des structures de contrôle qui peuvent exister dans un traitement itératif, en effet ce dernier peut avoir des énoncés conditionnels ainsi que des boucles imbriqués. L'accélération des boucles imbriqués pose un problème d'ordonnancement dit multidimensionnel. Ce champ est encore très peu exploré [WOL91, PAS94]. Le problème d'optimisation des connexions est particulier à la SHN et peut être primordial pour des circuits tels que les FPGAs. Le problème d'allocation de registres pose des défis particuliers différents du domaine de la compilation, on peut par exemple décider d'utiliser des files au lieu de banques de registres, ce qui nécessite moins de matériel tout en s'adaptant au traitement itératif [BEN96c, ALO94]. L'optimisation de l'interconnexion des différents modules est très importante.

Afin que la SHN fasse partie intégrante du processus de conception de circuits il faudra que chacun de ces problèmes soit résolu de manière satisfaisante pour le concepteur. Etant donné la difficulté inhérente à ces problèmes, les concepteurs recherchent aussi des algorithmes polynômiaux pour calculer des estimateurs qui les aiderait dans un processus de conception semi-automatique, où plusieurs alternatives peuvent être évaluées efficacement, ainsi les paramètres suivants sont intéressants à calculer:

- Borne supérieure de la fréquence réalisable étant donné un ensemble de ressources. L'équation (5) donne déjà une première borne.
- Borne inférieure de la latence réalisable pour une fréquence cible [HU93, BEN96a].
- Borne inférieure sur le nombre de registres nécessaires au stockage des variables étant donné un ordonnancement.

De tels algorithmes sont très utiles pour (1) explorer rapidement la performance et le coût de plusieurs réalisations matérielles d'une même application sous différentes contraintes de ressources, et (2) pour évaluer la qualité des solutions d'ordonnancement et d'allocation obtenues par les méthodes heuristiques.

Bien que les problèmes d'ordonnancement cyclique et d'allocation des registres soient NP-complets, il est utile d'avoir des algorithmes pour les résoudre d'une façon optimale. En effet, en pratique ces algorithmes prennent des temps d'exécution raisonnables pour des problèmes de petite taille et peuvent servir comme étalon pour des heuristiques.

Chapitre 2.

Méthode de calcul de la performance maximale d'un circuit itératif sous des contraintes de ressources

Résumé— La performance d'un circuit itératif est mesurée par trois valeurs: la durée du cycle d'horloge, la fréquence et la latence. La fréquence est égale au nombre moyen d'itérations exécutées par cycle d'horloge. Plus la fréquence d'un circuit est élevée plus sa performance est élevée. La latence est égale au nombre de cycles d'horloge nécessaires pour exécuter une itération. Dans certaines applications, la latence est un paramètre qu'il faut minimiser.

Dans ce chapitre, nous présentons des méthodes algorithmiques pour résoudre les deux problèmes suivants:

(1) Etant donnés un traitement itératif et un nombre limite de ressources, déterminer une borne supérieure sur la fréquence réalisable.

(2) Etant donnés un traitement itératif, un nombre limite de ressources, et une fréquence cible, déterminer une borne inférieure sur la latence réalisable.

Les travaux précédents ont traité seulement des cas particuliers de ces deux problèmes: le cas où il n'y a pas des contraintes de ressources, et le cas où il n'y a pas des dépendances de données entre les itérations.

Les méthodes que nous avons développées se basent sur la technique de relaxation de contraintes et ont une complexité polynomiale. Nous prouvons que cette complexité est minimale dans le cas du problème (2). Ces méthodes ont été testées sur des circuits de test fréquemment utilisés dans la synthèse de haut-niveau.

Le reste de ce chapitre est constitué de l'article [BEN 96a].

Lower Bounds on the Iteration Time and the Initiation Interval of Functional Pipelining and Loop Folding

Imed Eddine Bennour et El Mostapha Aboulhamid
Département d'informatique et recherche opérationnelle, université de Montréal
CP 6128, Centre ville, H3C-3J7,PQ, Canada

Abstract—The performance of pipelined datapath implementations is measured basically by three parameters: the clock cycle length, the initiation interval between successive iterations (inverse of the throughput) and the iteration time (turn-around time). In this paper we present a new method for computing performance bounds of pipelined implementations:

- Given an iterative behavior, a set of resource constraints and a target initiation interval, we derive a lower bound on the iteration time achievable by any pipelined implementation.
- Given an iterative behavior and a set of resource constraints, we derive a lower bound on the initiation interval achievable by any pipelined implementation.

The method has a low complexity and it handles behavioral specifications containing loop statements with inter-iteration data dependency and timing constrains.

2.1. Introduction

High level synthesis refers to the design process which transforms a behavioral specification of a digital system into a register transfer level (RTL) structure. Two fundamental steps in high level synthesis are scheduling and allocation. Scheduling assigns circuit operations to control steps under resource constraints (e.g. functional units, registers and buses) and/or performance constraint, while allocation assigns operations and data transfers to resources to realize the datapath. Synthesis of efficient circuits for real-time digital signal processing (DSP) applications is becoming a more challenging and crucial task, because most applications require higher sample rates and higher sample processing speed. These applications are often recursive or iterative and their behavioral

descriptions consist of an infinite loop statement. In order to synthesize a high throughput circuit, a scheduler should exploit all the potential concurrency between the loop body operations. A way to exploit this parallelism is to *pipeline* (overlap) the execution of successive iterations. This technique is called *loop folding* in the general case, and it is called *functional pipelining* (functional pipelined datapath) when there are no data dependencies between different iterations of the algorithm. Figure 2.1 illustrates the pipelining approach where a loop instance considered as a task is split into five subtasks ST_i ($i = 1, ..., 5$). Each subtask corresponds to a set of operations executed in parallel.

The performance of a pipelined datapath is measured basically by three indicators [PAR88]: the clock cycle length, the initiation interval and the iteration time. The *initiation interval* corresponds to the number of clock cycles separating the initiation of successive instances of the loop, it is the inverse of the throughput. The *iteration time* corresponds to the number of clock cycles necessary to execute one instance inside the pipeline (turn-around time), it measures the sample processing speed which is usually critical.

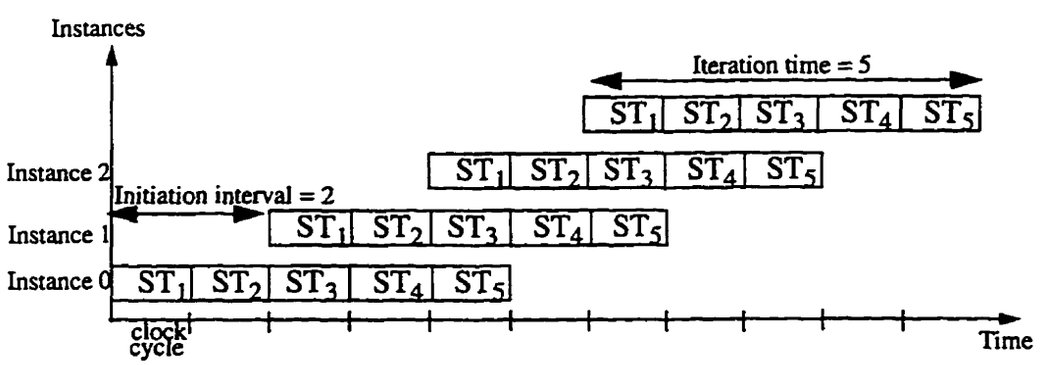


Figure 2.1 Space-time diagram of the pipelined schedule

The problem of determining the optimum initiation interval and the problem of determining the optimum iteration time for a given initiation interval are both NP-complete [GRO92] when there are resource constraints and inter-iteration data dependencies. The main objective of pipelined scheduling heuristics developed in the literature [HWA93, GOO90, WAN 93] is to find schedules with low initiation interval

and low iteration time.

In this paper we address the two following lower-bound problems:

Problem 1: Given a cyclic precedence graph representing a loop statement, a set of resource constraints and a target initiation interval, we derive a lower bound on the iteration time achievable by any pipelined implementation.

Problem 2: Given a cyclic precedence graph and a set of resource constraints, we derive a lower bound on the initiation interval achievable by any pipelined implementation.

2.2. Motivations and previous works

The motivations for developing performance estimators in high-level synthesis area have been discussed extensively in [RAB94, RIM 94, JAI 92]. Here we present three main motivations related to pipelined designs:

• Speed-up the space solution exploration. To achieve high performance pipelined designs, behavioral optimizations and dataflow-based transformations such as common subexpression elimination, associativity-commutativity algebraic transformations [POT94a, POT94b, SRI94, CHE91], loop unfolding [PAR91, LUC93], and retiming [LEI91] are often necessary. To find the best set of transformations and the best order requires the analysis of a large number of solutions. An efficient method to compute exact lower-bound performance allows to speed-up this exploration by detecting and removing solutions which theoretically cannot achieve the target performance under a given resource constraints.

• Evaluation of the quality of a pipelined solution produced by a heuristic. By comparing the exact lower-bound performance to the performance of the heuristic solution, we get the maximal difference between the heuristic solution and the optimum solution. A small difference indicates the good quality of the heuristic solution.

• Performance improvement of scheduling heuristics. The general framework used by

resource-constraint pipelined scheduling heuristics [LEE94, PAR93, GRO92] is composed of the following steps:

1. Fix the initiation interval to its tight lower bound;
2. For the target initiation interval, fix the iteration time to its tight lower bound;
3. Find a pipelined schedule with the current initiation interval and iteration time;
4. If no feasible schedule found and no time out, then increment the iteration time, and go to Step 3. Otherwise, increment the initiation interval, and go to Step 2.

Step3 is the most time consuming step since it is repeated many times and it is NP-complete. The use of tight lower bounds in steps 1 and 2 reduces this time considerably.

The majority of previous studies related to lower-bound performance estimation are restricted to non-pipelined designs [SHA93, RAB94, LAN93, RIM94, CHA94], and only few works have dealt with pipelined designs [JAIN92, HU93]. Jain *et al.* [JAIN92] have addressed the problem of area-delay prediction in pipelined and non-pipelined designs. For pipelined designs, they have presented a simple algorithm to compute a lower bound on the product of functional unit requirements with the initiation interval for acyclic data flow graphs. Although a lower bound on the product of these two parameters (number of functional units and initiation interval) permits to make design space exploration, this exploration is still restricted compared to the case where a lower bound on each parameter is given separately. Hu *et al.* [HU93] have presented a method to compute lower bounds on the iteration time under resource constraints but it is also restricted to acyclic data flow graphs. In [POT94], Potkonjak and Rabaey studied the relation between retiming and functional pipelining, and they gave a new polynomial algorithm for determining an upper bound on the throughput. However, the algorithm does not take into account the resource constraints. Other polynomial algorithms have been proposed [GER92, ZAK89] for determining an upper bound on the throughput and computing optimum rate schedules but they assume an unlimited number of resources.

Compared to the above mentioned works, the method presented here differs in both the scope and the techniques used. It computes lower bounds on the iteration time and on the initiation interval for an iterative behavior under resource constraints. The iterative

behavior may contain inter-iteration and intra-iteration precedence constraints, and user-specified min-max timing constraints between pairs of operations. It may also contain bounded nested loops; in such case all inner loops will be totally unfolded. The used technique is based on integer programming constraint relaxations; the proof that this technique has a minimal complexity is established. The method solves also the dual problem: lower bound on the resource requirement to achieve a target iteration time and/ or a target initiation interval. In this paper we restrict resource type to functional units. Registers and buses can be handled similarly by adding to the data-flow graph a set of fictive operations representing data memorizations and data transfers.

The rest of this paper is organized in the following way. Section 2.3 gives the necessary background. Section 2.4 describes the constraint graph model. Section 2.5 gives the general formulation for the minimum iteration time problem, then a relaxation of this problem, and the proof that the relaxed problem can be solved optimally in polynomial time by the proposed method. Lower-bound algorithms for the iteration time and for the initiation interval are given in Section 2.6 and Section 2.7, respectively. Experimental results are presented in Section 2.8.

2.3. Background

Cyclic precedence graph (CPG) is a well known model used to capture loop behavior. A CPG $G = (O, E, H, D)$ is a node-weighted and edge-weighted directed graph, where $O = \{o_i\}$ is the set of nodes representing atomic operations, E is the set of edges. An edge $(o_i \rightarrow o_j)$ corresponds to data precedence relationship between operations o_i and o_j. The integer edge weight $H(o_i \rightarrow o_j)$ means that the data produced by operation o_i in any iteration k of the loop will be used by operation o_j in iteration $(k + H(o_i \rightarrow o_j))$. The value $D(o_i)$ is the duration of operation o_i in clock cycles. Figure 2.2.a shows a CPG example, where normal arcs correspond to intra-iteration precedence constraints and they have the implicit weight zero, and dashed arcs correspond to inter-iteration precedence constraints. This example will be used throughout the paper.

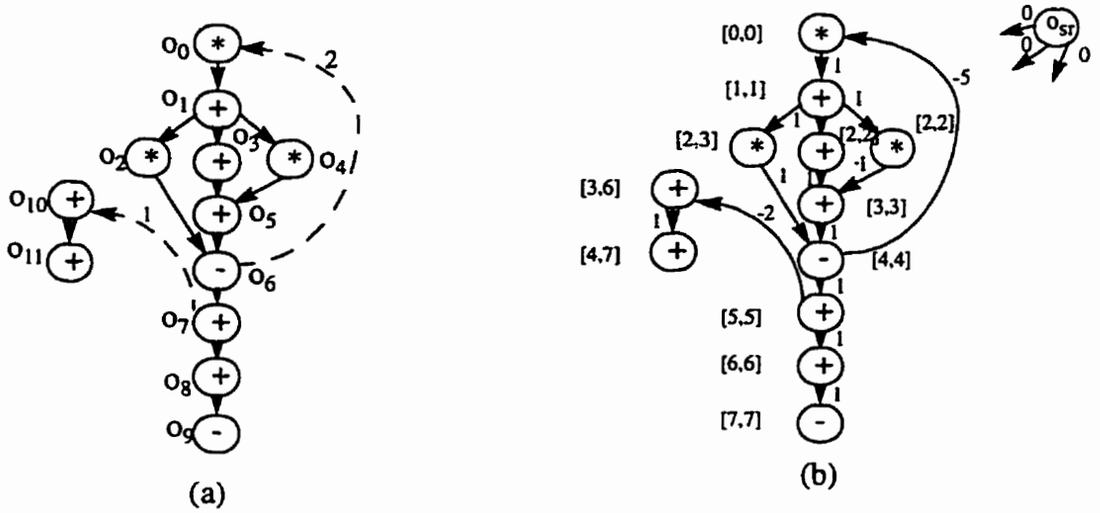


Figure 2.2 (a) A cyclic precedence graph example. (b) The constraint graph corresponding to $II = 3$, and the operation mobility-intervals.

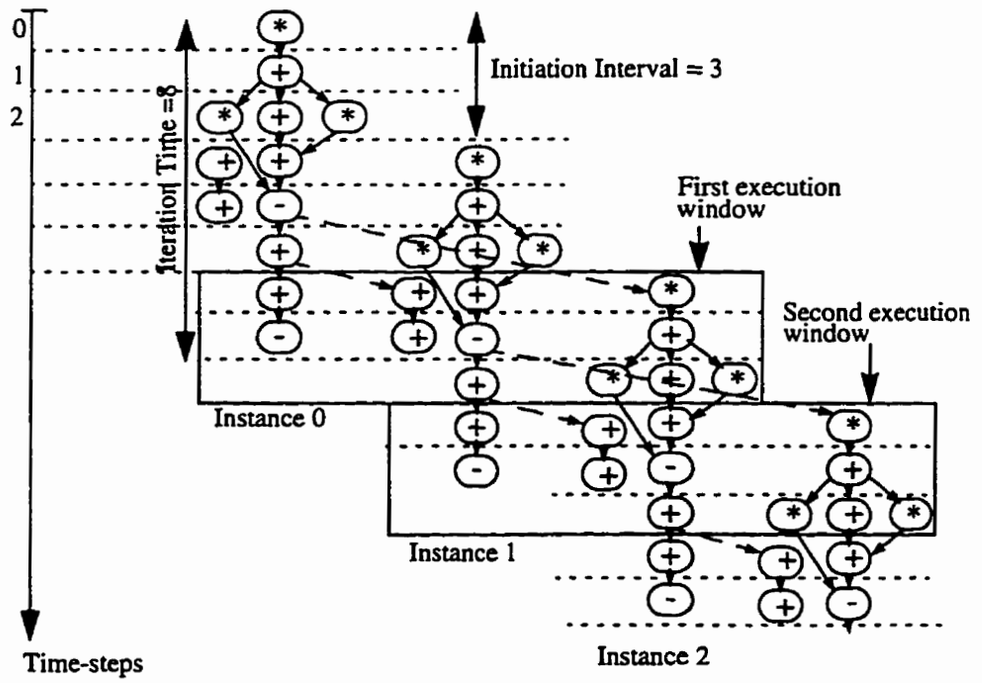


Figure 2.3 A pipelined schedule of the cyclic precedence graph of Figure 2.2.a.

Definition: Let $s(o_j, k)$ be the starting execution time of operation o_j of the k-th iteration of the loop. The schedule s is said to be a *pipelined schedule* with an initiation interval II if it satisfies:

$$\forall o_j \in O, \forall k \in \text{N (the set of natural numbers)} \cdot s(o_j, k) = s(o_j, 0) + k \cdot II \quad (1)$$

$$\text{and } \forall (o_i \rightarrow o_j) \in E \cdot s(o_j, H(o_i \rightarrow o_j)) \geq s(o_i, 0) + D(o_i) \quad (2)$$

Equation (1) expresses the schedule periodicity, and Equation (2) expresses intra-iteration and inter-iteration precedence constraints between operations. The iteration time (IT) of the schedule s is defined as:

$$IT = \max_{o_i} \{s(o_i, 0) + D(o_i)\} + 1 \quad (3)$$

As all iterations have the same schedule, they have the same iteration time. Figure 2.3 shows a pipelined schedule of the cyclic precedence graph (CPG) of Figure 2.2.a.

Initial lower bound on the initiation interval: The presence of loops in a CPG, called *loop carried dependencies* (LCD), limits the minimal value of the initiation interval. The lower bound value due to LCD, denoted II_{LDC}, is defined as follows [CHR83]:

$$II_{LDC} = \max_{C_k} \left\{ \left\lceil \frac{\sum_{o_i \in C_k} D(o_i)}{\sum_{(o_i \rightarrow o_j) \in C_k} H(o_i \rightarrow o_j)} \right\rceil \right\} \quad (4)$$

where C_k is a loop in the CPG.

Resource constraints also limit the minimal value of the initiation interval. This minimal value due to resource constraints, denoted II_{RC}, is given by [PAR88]:

$$II_{RC} = \max_{r} \left\{ \left\lceil \frac{\sum_{o_i \in T_r} D(o_i)}{M_r} \right\rceil \right\} \quad (5)$$

where r is a functional unit type, M_r is the available number of functional units of

type r, and T_r is the set of operations in the CPG executed by functional units of type r.

Theorem 1: *The initiation interval II of a feasible pipelined schedule must satisfy:*

$$II \geq \max(II_{RC}, II_{LDC})$$

2.4. The Constraint graph

Both intra-iteration and inter-iteration precedence constraints in a CPG will be transformed into timing constraints between operations of the *first loop iteration.* Combining Equations (1) and (2), we obtain:

$$\forall (o_i \rightarrow o_j) \in E.\ s(o_j, 0) \geq s(o_i, 0) + D(o_i) - H(o_i \rightarrow o_j) \cdot II \qquad (6)$$

Equation (6) expresses timing constraint between operations of the same first loop iteration.

Definition: Let $G = (O, E, H, D)$ be a cyclic precedence graph (CPG). The *constraint graph CG* associated to G is an edge-weighted and directed graph defined by $CG = (O \cup \{o_{sr}\}, E \cup E_{sr}, \hat{w})$, where O and E are the same sets of nodes and edges as in G, and o_{sr} is a source node connected to all nodes in O by the set of edges E_{sr}. The weights of edges $\hat{w}$ are defined as follows:

$$\forall (o_i \rightarrow o_j) \in E.\ \hat{w}(o_i \rightarrow o_j) = D(o_i) - H(o_i \rightarrow o_j) \cdot II$$

$$\forall (o_{sr} \rightarrow o_i) \in E_{sr}.\ \hat{w}(o_{sr} \rightarrow o_i) = 0$$

A positive (resp. negative) value of $\hat{w}(o_i \rightarrow o_j)$ means that operation o_j of the first iteration must be scheduled no sooner than $|\hat{w}(o_i \rightarrow o_j)|$ time-steps after (resp. before) operation o_i of the first iteration. Figure 2.2.b shows the constraint graph corresponding to the CPG of Figure 2.2.a for an initiation interval equal to 3. In the rest of the paper, we will use the constraint graph model instead of the CPG, and we assume that all multi-cyclic operations are broken into multiple uni-cycle operations related by timing constraints.

We denote by τ_i^s the "as soon as possible" starting time of operation o_i in the first iteration. The value of τ_i^s corresponds to the longest-path weight in the constraint graph from the source node o_{sr} to node o_i. The minimal number of time steps necessary to execute any iteration inside the pipeline corresponds to the critical path length in the constraint graph, denoted CP, and it is equal to:

$$CP = \max_i \{\tau_i^s + D(o_i)\} \tag{7}$$

We denote by τ_i^l the "as late as possible" starting time of operation o_i of the first loop instance such that the schedule length does not exceed CP. The interval $[\tau_i^s, \tau_i^l]$ is called *mobility-interval* of o_i.

In addition to data precedence constraints, the input circuit specification may contain timing constraints between pairs of operations. This second type of constraints can be represented directly in constraint graph.

2.5. ILP formulation of the minimum iteration time problem and its relaxation

In this section, we give first an ILP formulation for the *minimum iteration time problem*: find the minimum value of the iteration time, given a constraint graph, a fixed number of functional units and a target initiation interval. Then we construct a relaxation of this problem and we prove that the relaxed problem can be solved optimally in polynomial time by a greedy algorithm.

Notations:

- $GC = (O \cup \{o_{sr}\}, E \cup E_{sr}, \hat{w})$ is a constraint graph,
- τ_i^s is the earliest starting time of operation o_i of the first iteration, without resource constraints,
- τ_i^l is the latest starting time of operation o_i of the first iteration, without resource constraints,
- $M = \{1, 2, \ldots, m\}$ is the set of types of functional units,

- M_r is the number of functional units of type r,
- $T_r \subset O$ is the set of operations to be executed by functional units of type r,
- CP is the critical path length in the constraint graph,
- II is the initiation interval,
- IT is the iteration time,
- z is the additional number of time steps necessary to execute any loop iteration inside the pipeline; $z = IT - CP$.

2.5.1. *ILP formulation of the minimum iteration time problem* (P)

The system P formulates this NP-complete problem. A similar formulation is given in [HWA91].

$$P\begin{cases} \text{Minimize } z \text{ subject to:} & & (8) \\ \sum_{t=0}^{IT-1} x_{i,t} = 1 & \forall o_i \in O & (9) \\ \sum_{k=0}^{\lfloor (IT-e-1)/II \rfloor} \sum_{o_i \in T_r} x_{i,e+k\cdot II} \le M_r & \forall e \in \{0, 1, \ldots, II-1\}, \forall r \in M & (10) \\ \tau_j - \tau_i \ge \hat{w}(o_i \rightarrow o_j) & \forall (o_i \rightarrow o_j) \in E & (11) \\ \tau_i < CP + z & \forall o_i \in O & (12) \\ \text{where } \tau_i = \sum_{t=0}^{IT-1} t x_{i,t} & & \end{cases}$$

$x_{i,t}$ is a binary variable equal to 1 if operation o_i of the first iteration is scheduled in control step t, zero otherwise. The objective function (8) states that we minimize the iteration time $IT = CP + z$ of the schedule. Constraint (9) states that each operation o_i should be scheduled. Constraint (10) expresses the resource constraints: due to the pipelining, operations in control steps $e + k \cdot II$, for $k = 0, 1, 2, \ldots$, are executed simultaneously and cannot share the same functional units (see Figure 2.4.a). Constraint (11) expresses the timing constraints between operations. Constraint (12) states that all operations must be scheduled before control-step $(CP + z)$.

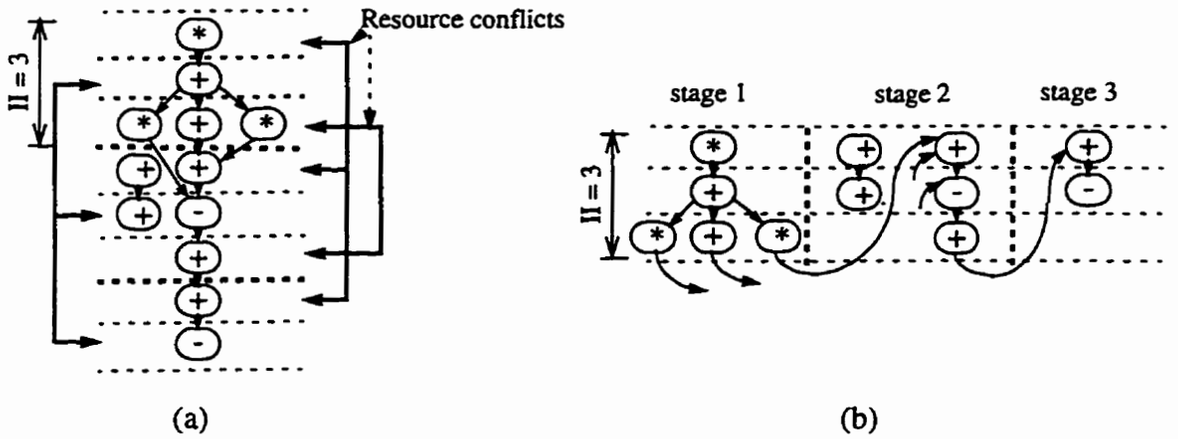


Figure 2.4 Two equivalent views of a pipelined schedule: (a) Single-iteration view, (b) Execution-window view.

2.5.2. The minimum iteration time relaxed problem

2.5.2.1. The relaxed problem ($\hat{PR}$)

The relaxation consists of replacing constraint (11) by a less strong constraint. This constraint is formalized by Equation (13). It defines the control-steps where operations must be scheduled: due to timing constraints between operations, an operation o_i cannot start before τ_i^s and must start no later than $\tau_i^l + z$. The relaxation problem is:

$$\hat{PR}\begin{cases} \text{Minimize } z \text{ subject to:} & \\ \sum_{t=0}^{IT-1} x_{i,t} = 1 & \forall o_i \in O \\ \sum_{k=0}^{\lfloor (IT-e-1)/II \rfloor} \sum_{o_i \in T_r} x_{i,e+k\cdot II} \le M_r & \forall e \in \{0,1,\ldots,II-1\}, \forall r \in M \\ \tau_i^s \le \tau_i \le \tau_i^l + z & \forall o_i \in O \\ \tau_i < CP + z & \forall o_i \in O \\ \text{where } \tau_i = \sum_{t=0}^{IT-1} t x_{i,t} & \end{cases} \quad (13)$$

2.5.2.2. Formulation of the relaxation problem based on execution-window's properties (PR)

The *execution-window* of a pipelined schedule is the repetitive pattern of the same size as the initiation interval that appears after a certain number of scheduling steps. In this section we give a second formulation of the relaxation problem $\hat{PR}$ based on the execution-window' properties.

The execution window starting at control step number $((\lceil IT/II \rceil - 1) \cdot II)$ is called the *first execution-window* (see Figure 2.3). A feasible pipelined schedule should satisfy the following three necessary conditions:

C1. Each operation o_i occurs one and only one time inside the first execution window.

C2. If an operation o_i of the first iteration is scheduled into control-step τ_i, then there is one occurrence of o_i scheduled into control-step $(\tau_i \ mod \ II)$ relatively to the beginning of the first execution window (see Figure 2.4). More generally, if o_i of the first iteration is scheduled inside the control-step interval $[\tau_i^s, \tau_i^l + z]$, for any positive integer z, then there is one occurrence of o_i scheduled, relatively to the beginning of the first window, inside the domain $DO_i(z)$ defined by:

$$DO_i(z) = \begin{cases} [0, II-1], & \text{if } (\tau_i^l + z - \tau_i^s) \geq II - 1 \\ [\tau_i^s \ mod \ II, \tau_i^s \ mod \ II + (\tau_i^l + z - \tau_i^s)] & \\ \quad \text{if } (\tau_i^s \ mod \ II + \tau_i^l + z - \tau_i^s) \leq II - 1 & \\ [\tau_i^s mod \ II, II-1] \cup [0, (\tau_i^l + z) \ mod \ II], & \text{otherwise} \end{cases} \quad (14)$$

The values $DO_i(0)$ for the illustrative example are given in Table 2.1 and represented in Figure 2.5.a.

C3. Resource constraints must be satisfied for every control step in the first execution window.

Table 2.1: Initial domains of operations for $II = 3$

	o_0	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
$[\tau_i^s, \tau_i^l]$	[0, 0]	[1, 1]	[2, 3]	[2, 2]	[2, 2]	[3, 3]	[4, 4]	[5, 5]	[6, 6]
$DO_i(0)$	[0, 0]	[1, 1]	[0, 0] ∪ [2, 2]	[2, 2]	[2, 2]	[0, 0]	[1, 1]	[2, 2]	[0, 0]

	o_9	o_{10}	o_{11}
$[\tau_i^s, \tau_i^l]$	[7, 7]	[3, 6]	[4, 7]
$DO_i(0)$	[1, 1]	[0, 2]	[0, 2]

The following system PR is equivalent to $\hat{PR}$ and it formalizes the conditions C1, C2 and C3.

$$PR: \begin{cases} \text{Minimize } z \text{ subject to:} & (15) \\ \sum_{t=0}^{II-1} y_{i,t} = 1 \qquad \forall o_i \in O & (16) \\ \sum_{t=0}^{II-1} t y_{i,t} \in DO_i(z) \qquad \forall o_i \in O & (17) \\ \sum_{o_i \in T_r} y_{i,t} \leq M_r \qquad \forall t \in \{0, \ldots, II-1\}, \forall r \in M & (18) \end{cases}$$

where $y_{i,t}$ equals to 1 if an occurrence of operation o_i is scheduled into control step t, zero otherwise. The objective function (15) states that we minimize the iteration time $IT = CP + z$. Constraints (16), (17) and (18) correspond to C1, C2 and C3, respectively.

The system PR can be solved optimally in $O(|O|^3)$ using the sub-interval method [HU93]. We propose a new method which solves the same system optimally in $O(|O|^2)$. The next section presents the theoretical framework for this method.

2.5.3. *Transformation of PR to PRU(S)*

In the system *PR*, the initial scheduling domains $\{DO_i(0)\}$ of operations could be the union of two disjoint intervals (Equation 14). The goal of the proposed transformation is to construct a new scheduling problem *PRU(S)* equivalent to *PR* where the scheduling domains of operations are single intervals.

PRU(S) is obtained by generating for each operation $o_i \in O$ a number S of operations $o_{i,j}$, $j = 0, 1, \ldots, S-1$, and associating to $o_{i,j}$ an initial single interval domain $[\tau^s_{i,j}, \tau^l_{i,j}]$ defined by:

$$[\tau^s_{i,j}, \tau^l_{i,j}] = [\tau^s_{i,j-1} + II, \tau^l_{i,j-1} + II] \tag{19}$$

$$\text{with } [\tau^s_{i,0}, \tau^l_{i,0}] = \begin{cases} [\tau^s_i \bmod II, \tau^s_i \bmod II + (\tau^l_i - \tau^s_i)], & \text{if } (\tau^l_i - \tau^s_i) \le II - 1 \\ [\tau^s_i \bmod II, \tau^s_i \bmod II + II - 1], & \text{otherwise} \end{cases} \tag{20}$$

Figure 2.5 illustrates this unfolding like-transformation.

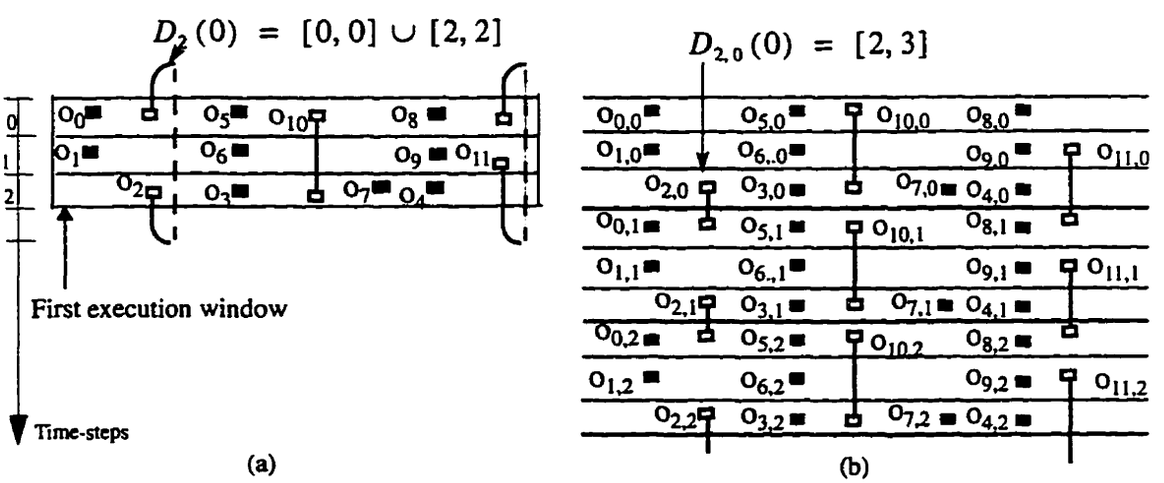


Figure 2.5 (a) Representation of the initial operation domains $DO_i(0)$, (b) Transformation of the scheduling problem represented in (a).

Let $O_S = \{o_{i,j} \ / o_i \in O, j = 0, 1, \ldots, S-1\}$. The ILP formulation for the parametrized system *PRU(S)* is:

$$PRU(S)\begin{cases}\text{Minimize } z \text{ subject to:} & (21)\\ \sum_{t=0}^{(S+1)II-1} y_{i,j,t} = 1 \qquad \forall o_{i,j} \in O_S & (22)\\ \tau^s_{i,j} \le \tau_{i,j} \le \tau^l_{i,j} + z \qquad \forall o_{i,j} \in O_S & (23)\\ \sum_{o_{i,j} \in O_S / o_i \in T_r} y_{i,j,t} \le M_r \qquad \forall t \in \{0, 1, \ldots, (S+1)II-1\}, \forall r \in M & (24)\\ \text{where,} \\ \tau_{i,j} = \sum_{t=j \cdot II}^{(j+2)II-1} t y_{i,j,t}\end{cases}$$

$y_{i,j,t}$ equals 1 if operation $o_{i,j}$ is scheduled in control-step t, zero otherwise. For a fixed value of S, the system *PRU(S)* formulates the following scheduling problem: given a set of functional units $\{M_r\}$, a set of operations $O_S = \{o_{i,j}\}$ defined by their initial scheduling domains $\{[\tau^s_{i,j}, \tau^l_{i,j}]\}$, what is the minimal number of additional time-steps necessary to schedule all operations without resource conflict? It was proven in [RIM94] that this type of problems can be solved optimally in polynomial time by a greedy algorithm. This algorithm is given in appendix A.

Theorem 2: *For a fixed value of S, the system* $PRU(S)$ *can be solved optimally in polynomial time.*

The next theorem states that the systems $PRU(2)$ and $PRU(S)$ $\forall S \ge 2$ have the same optimum value of their objective function, and thus it is sufficient to solve $PRU(2)$.

Theorem 3: $\forall S \ge 2$, $PRU(S) \Leftrightarrow PRU(2)$

Proof: given in appendix B.

The following expression recapitulates the set of ILP transformations done in this section:

$$P \Rightarrow \hat{PR} \Leftrightarrow PR \Leftrightarrow PRU(2)$$

where $P \Rightarrow \hat{PR}$ means that $\hat{PR}$ is a relaxation of P, and $\hat{PR} \Leftrightarrow PR$ means that both systems are equivalent.

2.6. Lower bound on the iteration time algorithm

In this section we present the basic algorithm to compute lower bounds on the iteration time under resource constraints. This algorithm is based on the results of Theorems 2 and 3. We show then how the same algorithm is used to compute tight operation mobility-intervals.

2.6.1. Basic algorithm

Given a constraint graph *CG*, a target initiation interval II, and a set of functional units M_r, the algorithm given in Figure 2.6 computes a lower bound on the iteration time using three steps: steps 1 and 2 construct the system $PRU(2)$, and step 3 solves it optimally.

Example: We illustrate the *IT_LowerBound* algorithm on the graph of Figure 2.2.b under the constraint of one multiplier, one subtracter and four adders. This graph has a critical path length *CP* equal to 8. The sets $\{[\tau_i^s, \tau_i^l]\}$, $\{[\tau_{i,0}^s, \tau_{i,0}^l]\}$ and $\{[\tau_{i,1}^s, \tau_{i,1}^l]\}$ are given in Table 2.2. Sorting the operations by the increasing values of $\tau_{i,j}^l$, we obtain:

$$\{o_{0,0}, o_{5,0}, o_{8,0}, o_{1,0}, o_{6,0}, o_{9,0}, o_{3,0}, o_{4,0}, o_{7,0}, o_{10,0}, o_{11,0}, o_{2,0}, o_{0,1}, o_{5,1}, o_{8,1}, o_{1,1}$$
$$o_{6,1}, o_{9,1}, o_{3,1}, o_{4,1}, o_{7,1}, o_{10,1}, o_{11,1}, o_{2,1}\}$$

As there is only one multiplier unit, it is not possible to schedule operations $o_{3,0}$, $o_{2,0}$ and $o_{0,1}$ inside their respective initial domains $[2, 2]$, $[2, 3]$ and $[3, 3]$, unless the critical path length is increased by one time-step. Similar resource conflicts occur between the multiplications $o_{6,0}$, $o_{9,0}$, $o_{6,1}$ and $o_{9,1}$ which have as initial domains $[1, 1]$, $[1, 1]$, $[4, 4]$ and $[4, 4]$, respectively. Thus, a lower bound on the iteration time will be $CP + 1 = 9$.

Algorithm: *IT_LowerBound*

input: constraint graph *CG*, target initiation interval *II*, set of resources $\{M_r\}$,

output: lower bound of the iteration time (IT_{LB})

begin

Step 1: Break multi-cycle operations into uni-cycle operations;

Compute the mobility $[\tau_i^s, \tau_i^l]$ of each operation o_i in *CG* using Bellman-Ford algorithm;

$CP = Max(\tau_i^s + 1)$;

Step 2: **for** each operation o_i **do**

Create two operations $o_{i,0}$ and $o_{i,1}$

for $j = 0, 1$ **do**

$$\tau_{i,j}^s = \tau_i^s \bmod II + j \cdot II \ ;$$

$$\tau_{i,j}^l = \begin{cases} \tau_i^s \bmod II + (\tau_i^l - \tau_i^s) + j \cdot II & \text{if } (\tau_i^l - \tau_i^s) < II - 1 \\ \tau_i^s \bmod II + (II - 1) + j \cdot II & \text{otherwise} \end{cases}$$

end for

end for

Step 3: Sort the set $\{o_{i,j}\}$ by the increasing values of $\tau_{i,j}^l$;

Assign each operation $o_{i,j}$ to the first control step $\tau_{i,j}$ after $\tau_{i,j}^s$ containing a free resource;

$z = max(\tau_{i,j} - \tau_{i,j}^l)$;

$IT_{LB} = z + CP$;

end

Figure 2.6 Algorithm to find a lower bound of the iteration time.

Table 2.2: Intermediate results of the *IT_LowerBound* algorithm

	o_0	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
$[\tau_i^s, \tau_i^l]$	[0, 0]	[1, 1]	[2, 3]	[2, 2]	[2, 2]	[3, 3]	[4, 4]	[5, 5]	[6, 6]
$[\tau_{i,0}^s, \tau_{i,0}^l]$	[0, 0]	[1, 1]	[2, 3]	[2, 2]	[2, 2]	[0, 0]	[1, 1]	[2, 2]	[0, 0]
$[\tau_{i,1}^s, \tau_{i,1}^l]$	[3, 3]	[4, 4]	[5, 6]	[5, 5]	[5, 5]	[3, 3]	[4, 4]	[5, 5]	[3, 3]

	o_9	o_{10}	o_{11}
$[\tau_i^s, \tau_i^l]$	[7, 7]	[3, 6]	[4, 7]
$[\tau_{i,0}^s, \tau_{i,0}^l]$	[1, 1]	[0, 2]	[0, 2]
$[\tau_{i,1}^s, \tau_{i,1}^l]$	[4, 4]	[3, 5]	[4, 5]

Algorithm complexity: Operation mobility-intervals are computed using Bellman-Ford algorithm [COR90] in $O(|O||E|)$, where O and E are the set of nodes and the set of arcs in the constraint graph, respectively. As an operation has in average two input arcs, the complexity of Bellman-Ford algorithm is in $O(|O|^2)$. The sorting and the greedy scheduling of $2 \cdot |O|$ operations can be done in $O(|O|^2)$. Thus, the algorithm complexity is $O(|O|^2)$. Notice that this complexity is minimal since computing the earliest starting times in a cyclic graph is in $O(|O|^2)$.

2.6.2. Operation mobility-intervals under pipelining and resource constraints

An important property of the *IT_LowerBound* algorithm (Figure 2.6) is that: the more accurate are the operation mobility-intervals $\{ [\tau_i^s, \tau_i^l] \}$, the tighter lower bound is returned. This section presents a method to compute more accurate operation mobility-intervals. The method is based on the technique introduced initially in [LAN93] for computing lower-bound performance of non-pipelined schedules in acyclic data flow graphs.

To obtain tight operation mobility-intervals, three types of constraints are considered simultaneously: timing constraints, resource constraints and pipelining. Let $\left[{}^{rc}\tau_i^{\,s}, {}^{rc}\tau_i^{\,l}\right]$ denote the mobility-interval of o_i under the three previous constraints. Let CG_{o_i} be a subgraph of the constraint graph CG containing operation o_i and all operations o_j in CG such that there exists at least one path in CG from o_j to o_i having only positive edges (see Figure 2.7.a). Tight value of ${}^{rc}\tau_i^{\,s}$ is obtained by performing the *IT_LowerBound* algorithm on CG_{o_i}: as operation o_i is always the latest operation executed among operations in CG_{o_i}, a lower bound on the iteration time of CG_{o_i} constitutes an earliest starting time of o_i.

The order of computing $\{{}^{rc}\tau_i^{\,s}\}$ is important, because it influences their final values. An efficient order consists of computing ${}^{rc}\tau_i^{\,s}$ only after computing all ${}^{rc}\tau_j^{\,s}$ such that $\hat{w}(o_j \rightarrow o_i) > 0$.

By inverting the direction of edges in CG, the same technique allows to compute $\{{}^{rc}\tau_i^{\,l}\}$. Figure 2.7.b shows the accuracy improvement of mobility-intervals for the illustrative example. Notice that the lower bound on the iteration of the whole graph is also improved from 9 to 10. The use of tight operation mobility-intervals reduces automatically the running time of both exact and heuristic schedulers.

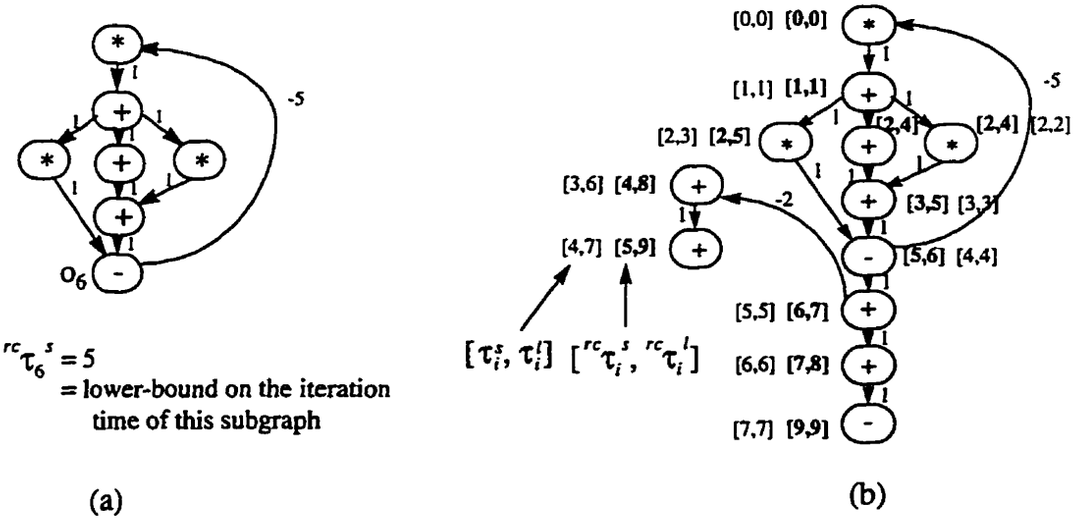


*Figure 2.7 (a) Subgraph CG_{o_6}. (b) Operation mobility-intervals under pipelining and resource constraints (4 +, 1 *, 1 —).*

The algorithm to compute tight earliest starting times of operations is given in Figure 2.8. Its complexity is $O(|O|^3)$, since for each subgraph CG_{o_i} we perform the *IT_LowerBound* algorithm which is in $O(|O|^2)$. The same algorithm can be used for compute tight lateast starting times of operations by reversing the direction of arcs in the constraint graph.

2.7. Lower bound on the initiation interval algorithm

The description of the algorithm to find a lower bound on the initiation interval is given in Figure 2.9. Starting from the minimal value given by Theorem 1, the initiation interval will be increased iteratively as long as the current value induces timing-constraint inconsistencies in the constraint graph *CG*. For a target initiation interval, timing-constraint consistencies are checked as follows: for each arc $(o_i \rightarrow o_j)$ in *CG* having a negative weight $(\hat{w}(o_i \rightarrow o_j) < 0)$, we compute a lower bound on the number of control steps separating operation o_i from o_j in any feasible pipelined schedule. This lower bound is obtained by performing the *IT_LowerBound* algorithm on a specific subgraph $CG_{o_i \rightarrow o_j}$ of GC. The subgraph $CG_{o_i \rightarrow o_j}$ contains operations o_i, o_j and all operations o_k such that o_k belongs to at least one path from o_j to o_i in *CG* having only positive weights assigned to edges. If the obtained lower bound is greater than $|\hat{w}(o_i \rightarrow o_j)|$, then the current initiation interval is not feasible and it is increased, otherwise this lower bound will be the new weight of the edge $(o_j \rightarrow o_i)$.

Each pass of the algorithm has a complexity of $O(m \cdot |O|^2)$, where m is the number of negative arcs in the constraint graph.

Algorithm: *Minimum_starting_time*

Input: constraint graph *CG*, initiation interval *II*, set of resources $\{M_r\}$

Output: earliest starting times $\{{}^{rc}\tau_i{}^{s}\}$ of operations in *CG*

begin

Step 1: **for** each operation o_i in *CG* **do**

$${}^{rc}\tau_i{}^{s} = \text{earliest starting time of } o_i \text{ by resolving the longest-path problem;}$$

$$\hat{w}(o_{sr} \rightarrow o_i) = {}^{rc}\tau_i{}^{s};$$

end for

Step 2: **for** each operation o_i such that $\hat{w}(o_i \rightarrow o_j) < 0, \forall (o_i \rightarrow o_j)$ **do**

$${}^{rc}\tau_i{}^{s} = \textit{Starting\_time\_constraints}\ (o_i);$$

end for

end

Function: *Starting_time_constraints*(o_i)

begin

Step 2-0: **if** this function has already been executed on operation o_i **then** return ${}^{rc}\tau_i{}^{s}$;

else

Step 2-1: **for** each operation o_j such that $\hat{w}(o_j \rightarrow o_i) > 0$ **do**

$${}^{rc}\tau_j{}^{s} = \textit{starting\_time\_constraints}\ (o_j);$$

$$\hat{w}(o_{sr} \rightarrow o_j) = {}^{rc}\tau_j{}^{s}; \quad \text{/* update arcs' weights in } CG \text{ */}$$

end for

Step 2-2: Recompute the earliest starting time ${}^{rc}\tau_k{}^{s}$ of all nodes in *CG* by resolving the longest-path problem in *CG;*

$$\hat{w}(o_{sr} \rightarrow o_k) = {}^{rc}\tau_k{}^{s};$$

Step 2-3: Return (*IT_LowerBound* (CG_{o_i}, *II*, $\{M_r\}$));

end if

end

Figure 2.8 Algorithm to compute earliest starting times of operations under pipelining, timing and resource constraints.

Algorithm: *II_LowerBound*
input: constraint graph *CG*, set of resources $\{M_r\}$
output: lower bound on the initiation interval (II_{LB})

begin

$II_{LB} = Max\{II_{RC}, II_{LDC}\}$;

$SG = \{CG_{o_i \rightarrow o_j} \mid \hat{w}(o_i \rightarrow o_j) < 0\}$;

while there are timing-constraint inconsistencies **do**

for each subgraph $CG_{o_i \rightarrow o_j}$ of the set *SG* **do**

$IT = IT\_LowerBound\,(CG_{o_i \rightarrow o_j}, II_{LB}, \{M_l\})$;
if $IT > |\hat{w}(o_i \rightarrow o_j)|$ **then** /* timing constraint inconsistency*/
$II_{LB} = II_{LB} + 1$;
else
$\hat{w}(o_j \rightarrow o_i) = IT$;
end for

end while
end

Figure 2.9 Algorithm to find a lower bound on the initiation interval.

2.8. Experimental results

We tested the estimation method on different high-level synthesis benchmarks. For each benchmark, lower bounds were computed and compared to realizable solutions obtained by the pipelined scheduling systems Theda/Fold [LEE94] and PLS [HWA93]. These realizable solutions are considered as upper bounds. Algorithms are implemented in the C language and run on a SPARC 10 station. Experimental results are shown in Tables 2.3 to 2.8, where the field *RC* indicates the number of resources, II_{UB} and IT_{UB} are, respectively, the upper bounds on the iteration time and the upper bound on the initiation interval published in [LEE94, HWA93], II_{LB} and IT_{LB} are the computed lower bounds. Unless it is specified, we assume that multipliers take two clock-cycles while adders and subtracters take one clock-cycle. ($*^P$) denotes multipliers with two stages pipelining.

The Fifth-order digital filter [KUN85]: this benchmark contains 26 additions and 8 multiplications, it has a large number of intra-iteration and inter-iteration precedence constraints and many loops carried dependencies (LCDs). The minimal initiation interval due to LCDs is equal to 16. Table 2.3 shows the experimental results for this benchmark.

The second-order IIR filter (Table 2.4): its cyclic precedence graph is shown in Figure 2.10.

The third-order IIR filter [JEN94] (Table 2.5): it contains 6 additions, 2 multiplications and 3 LCDs. The minimal initiation interval due to LCDs is equal to 3.

The 16-tap FIR filter [PAR88] (Table 2.6): it contains 15 additions, 8 multiplications and only intra-iteration data dependencies. We assume that an addition has a duration of 40ns, a multiplication has a duration of 80ns, the clock cycle length is equal to 100ns, and operation chaining are permitted.

The Fast Discrete Cosine Transformation (Table 2.7): This benchmark is relatively large, it contains 13 additions, 13 subtractions and 16 multiplications. The algorithmic description of this benchmark is a straight-line code, not a loop statement. But we can consider it as a loop statement if we assume that the algorithm is to be performed on many

sets of data.

The Fifth-order Digital Wave Filter with no LCDs (Table 2.8): It is the same benchmark as the first one where all the inter-iteration precedence constraints are removed.

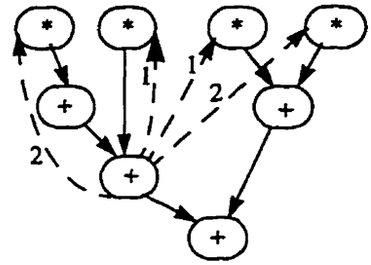


Figure 2.10 The second-order IIR filter.

Table 2.3: Fifth order digital wave filter

RC	Π_{UB}	IT_{UB}	Π_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
3 +, 3 *	16	18	16	17	1
3 +, 2 *	16	18	16	18	0
2 +, 2 *	17	19	16	18	1
2 +, 1 *	19	21	17	21	0
3 +, 2 $*^P$	16	18	16	17	1
3 +, 1 $*^P$	16	18	16	18	0
2 +, 1 $*^P$	17	19	17	18	1
1 +, 1 $*^P$	28	28	28	28	0

Table 2.4: Second-order IIR filter

RC	II_{UB}	IT_{UB}	II_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
2 +, 3 *	3	6	3	5	1
2 +, 2 *	4	6	4	6	0
1 +, 2 *	4	7	4	6	1
1 +, 1 *	8	10	8	10	0
2 +, 2 $*^P$	3	5	3	5	0
1 +, 2 $*^P$	4	6	4	6	0
1 +, 1 $*^P$	4	7	4	6	1

Table 2.5: Third-order IIR filter

RC	II_{UB}	IT_{UB}	II_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
2 +, 2 *	3	5	3	5	0
2 +, 1 *	4	6	4	6	0
1 +, 1 *	6	6	6	6	0
2 +, 1 $*^P$	3	5	3	5	0
1 +, 1 $*^P$	6	6	6	6	0

Table 2.6: 16-point FIR filter

RC	II_{UB}	IT_{UB}	II_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
15 +, 8 *	1	6	1	6	0
8 +, 4 *	2	6	2	6	0
6 +, 3 *	3	6	3	6	0
5 +, 3 *	3	6	3	6	0
4 +, 2 *	4	6	4	6	0
3 +, 2 *	5	7	5	6	1
2 +, 1 *	8	10	8	10	0
1 +, 1 *	15	16	15	15	1

Table 2.7: Fast discrete cosine transformation

RC	Π_{UB}	IT_{UB}	Π_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
5+, 5—, 6*^{P}	3	9	3	9	0
4+, 4—, 4*^{P}	4	10	4	9	1
3+, 3—, 4*^{P}	5	10	5	9	1
3+, 3—, 3*^{P}	6	11	6	9	2
2+, 2—, 2*^{P}	8	12	8	11	1
1+, 1—, 2*^{P}	13	16	13	15	1

Table 2.8: Fifth order digital wave filter with no LCD's

RC	Π_{UB}	IT_{UB}	Π_{LB}	IT_{LB}	$IT_{UB} - IT_{LB}$
26 +, 8 *^{P}	1	17	1	17	0
13 +, 4 *^{P}	2	17	2	17	0
9 +, 3 *^{P}	3	18	3	17	1
7 +, 2 *^{P}	4	19	4	18	1
6 +, 2 *^{P}	5	19	5	18	1
5 +, 2 *^{P}	6	17	6	17	0
4 +, 2 *^{P}	7	18	7	18	0
4 +, 1 *^{P}	8	20	8	20	0
3 +, 1 *^{P}	9	22	9	20	2

The last column in Tables 2.3 to 2.8 shows that the lower bounds on the iteration time (IT_{LB}) obtained are close to the upper-bounds (IT_{UB}): the worst results are within two times-steps from the upper bounds. The running time of the algorithm is less than 0.08 seconds for all benchmarks. These results illustrate that the relaxation done on the minimum iteration time problem which is NP-complete is efficient: it does not induce a

large accuracy lost, while using a polynomial execution time. For the initiation intervals, the obtained lower bounds (II_{LB}) are equal to the upper bounds (II_{UB}) except for two experimental cases shown in bold in table 2.3.

During the scheduling of a data-flow graph, lower bounds constitute good measures to evaluate the performance quality of a solution and to decide when to stop the research of new solutions. If the difference between the performance of the current solution and the lower bound is equal to zero then the solution is optimum. Otherwise, this difference indicates the maximal performance improvement that could be achieved by continuing the research (or the maximal performance loss by choosing such solution). For the above benchmarks, 58% of the scheduling solutions obtained by the heuristics [LEE94, HWA93] are optimum, 37% are at most one control step from the optimum, and 5% are at most two control steps from the optimum.

2.9. Conclusions

Fast algorithms for performance estimation allow to make efficient design space exploration and to improve the quality and the performance of pipelined scheduling heuristics. In this paper, we have presented a new method for computing lower bounds on the iteration time and on the initiation interval of pipelined datapath implementations under functional unit constraints. The method handles behavioral description containing loop statements and timing constraints. It can handle implementations with operation chaining, multicycle operations, and pipelined functional units. Based on an efficient relaxation of the general pipelined scheduling problem, we have developed lower-bound algorithms with complexities of ($O(|O|^2)$ and $O(m \cdot |O|^2)$). The dual problem of computing lower bound on the resource requirement to achieve a target performance can be solved using the same method. Applications of our approach are not limited to the area of high-level synthesis, it can be used in different areas of computer design where functional pipelining and loop folding are used.

Appendix A

$$PRU(S)\begin{cases}\text{Minimize } z \text{ subject to:} & (25)\\ \sum_{t=0}^{(S+1)II-1} y_{i,j,t} = 1 \qquad \forall o_{i,j} \in O_S & (26)\\ \tau^s_{i,j} \le \tau_{i,j} \le \overline{\tau^l_{i,j}} = \tau^l_{i,j} + z \qquad \forall o_{i,j} \in O_S & (27)\\ \sum_{o_{i,j} \in O_S / o_i \in T_r} y_{i,j,t} \le M_r \qquad \forall t \in \{0, 1, \ldots, (S+1)II-1\}, \forall r \in M & (28)\\ \text{where} & \\ \tau_{i,j} = \sum_{t=j\cdot II}^{(j+2)II-1} t y_{i,j,t} & (29)\\ \tau^s_{i,j} = \tau^s_{i,0} + j\cdot II = \tau^s_i \bmod II + j\cdot II & (30)\\ \tau^l_{i,j} = \tau^l_{i,0} + j\cdot II = \begin{cases}\tau^s_i \bmod II + (\tau^l_i - \tau^s_i) + j\cdot II & \text{if } (\tau^l_i - \tau^s_i) < II-1\\ \tau^s_i \bmod II + (II-1) + j\cdot II & \text{otherwise}\end{cases} & (31)\end{cases}$$

Algorithm: *Greedy-scheduling*

Input: instance of the system $PRU(S)$ with a fixed value of S, set of resources

output: optimal value of the objective function z of $PRU(S)$

begin

Sort operations $o_{i,j}$ by the increasing values of $\tau^l_{i,j}$;

Assign each operation $o_{i,j}$ to the first step $\tau_{i,j}$ after $\tau^s_{i,j}$ which contains a free resource;

$z = max(\tau_{i,j} - \tau^l_{i,j})$;

end

Appendix B

The following proposition is used in the proof of Theorem 3.

Proposition 1

(a) For any value of z in *PRU(S)*, the maximal number of operations $o_{i,j}$ of a type r such that all the values of $\overline{\tau_{i,j}^l}$ belong to a same integer interval of size II, is less or equal to $|T_r|$, where $|T_r|$ is by definition the number of operations $o_{i,0}$ of type r in *PRU(S)*.

(b) If there exist two operations $o_{i,j}$ and $o_{k,l}$ in the system *PRU(S)* such that the corresponding values $\tau_{i,j}^s$ and $\tau_{k,l}^s$ belong to a same integer interval of size II, then $|l-j| \leq 1$.

Proof:

(a) By Equations (27) and (31) we have:

$$\tau_{i,j} \leq \overline{\tau_{i,j}^l} = \tau_{i,j}^l + z \qquad \forall o_{i,j} \in O_S \tag{32}$$

$$\tau_{i,j+1}^l = \tau_{i,j}^l + II$$

$$\Rightarrow \qquad \tau_{i,j+1} \leq \overline{\tau_{i,j+1}^l} = \tau_{i,j}^l + II + z \tag{33}$$

From (32) and (33) we deduce:

$$\left|\overline{\tau_{i,j}^l} - \overline{\tau_{i,k}^l}\right| \geq II \qquad \forall j \neq k$$

Thus, for any integer interval of size II, it is not possible to have two operations $o_{i,j}$ and $o_{i,k}$ such that both values $\overline{\tau_{i,j}^l}$ and $\overline{\tau_{i,k}^l}$ belong to this interval. As by definition the number of operations $o_{i,j}$ of the same type r and for a fixed index j is equal to $|T_r|$, then the number of operations $o_{i,j}$ having $\overline{\tau_{i,j}^l}$ included in the same interval of size II is less or equal to $|T_r|$.

(b) From Equation (30) we have:

$$\tau^s_{i,j} = \tau^s_i \bmod II + j \cdot II < (j+1) \cdot II \qquad \forall o_{i,j} \in O_S$$

$$\Rightarrow \quad \tau^s_{k,j+2} = \tau^s_k \bmod II + (j+2) \cdot II \geq (j+2) \cdot II$$

$$\Rightarrow \quad \tau^s_{k,j+2} - \tau^s_{i,j} > II \tag{34}$$

From (34) we deduce that if there exist two operations $o_{i,j}$ and $o_{k,l}$ such that $\tau^s_{i,j}$ and $\tau^s_{k,l}$ belong to a same interval of size II, then $|l-j| \leq 1$. ❑

Theorem 3: $\forall S \geq 2$, $PRU(S) \Leftrightarrow PRU(2)$

Proof:

The proof is by contradiction. Suppose that the optimum value of the objective function of $PRU(2)$ is equal to z_0, and that z_0 is also an optimum solution of $PRU(n)$ for $n \geq 2$ but not for $PRU(n+1)$. This implies that if we fix z to z_0 in the constraint (27) of the system $PRU(S = n+1)$, and we use the greedy algorithm (given in appendix A) to schedule operations $o_{i,j}$ under resource constraints, then no feasible solution can be found. We will prove that such case cannot happen.

Let be $o_{g,h}$ the first operation which can not be scheduled before its latest starting time $\overline{\tau^l_{g,h}}$ due to the lack of resources, and let r the type of $o_{g,h}$. Depending on the number of resources already used between the control steps $(\overline{\tau^l_{g,h}} - II + 1)$ and $\overline{\tau^l_{g,h}}$, we distinguish two cases.

Case 1 In every control step of the interval $\left[\overline{\tau^l_{g,h}} - II + 1, \overline{\tau^l_{g,h}}\right]$ all the M_r resources are used. Since by Theorem 1, the value of II must satisfy $M_r \cdot II \geq |T_r|$, we deduce that there are at least $|T_r|$ operations already scheduled in this interval. Thus, there are at least $|T_r| + 1$ operations $o_{i,j}$, including the operation $o_{g,h}$, such that their values $\overline{\tau^l_{i,j}}$ belong to the interval $\left[\overline{\tau^l_{g,h}} - II + 1, \overline{\tau^l_{g,h}}\right]$. This is in contradiction with Proposition 1.(a)

Case 2 There is at least one control step p in $\left[\overline{\tau^l_{g,h}} - II + 1, \overline{\tau^l_{g,h}}\right]$ which contains free resources. As we use the greedy scheduling technique, all operations scheduled after the

control step p have the values of $\tau^s_{i,j}$ strictly greater than p. Let A be the set of operations $o_{i,j}$ already scheduled inside the interval $[p+1, \overline{\tau^l_{g,h}}]$. Since it was not possible to schedule operation $o_{g,h}$ before the control step $\overline{\tau^l_{g,h}}$, it implies that the following system SS does not admit a solution:

$$SS: \begin{cases} \sum_{t=p+1}^{\overline{\tau^l_{g,h}}} y_{i,j,t} = 1 & \forall o_{i,j} \in A \cup \{o_{g,h}\} \\ \sum_{o_{i,j} \in A \cup \{o_{g,h}\} / o_i \in T_r} y_{i,j,t} \leq M_r & \forall t \in \{p+1, \ldots, \overline{\tau^l_{g,h}}\}, r \in M \\ p+1 \leq \tau^s_{i,j} \leq \tau_{i,j} \leq \tau^l_{i,j} + z_0 & \forall o_{i,j} \in A \cup \{o_{g,h}\} \end{cases}$$

where $\tau_{i,j}$, $\tau^s_{i,j}$ and the set of constraints have the same meaning as in the system *PRU(S)*. As the size of the interval $[p+1, \overline{\tau^l_{g,h}}]$ is less than II, from Proposition 1.(b) we deduce that for any pair of operations $(o_{i,j}, o_{k,l})$ which belong to the set $A \cup \{o_{g,h}\}$ we have $|l-j| \leq 1$. By substituting in the system SS the smaller index j by 0 and the index $j+1$ by 1, we obtain an equivalent system which is completely included in the system $PRU(2)$. This is in contradiction with the hypothesis that $PRU(2)$ has a solution. ❑

Chapitre 3.

Méthode d'analyse statique pour estimer la performance d'un programme sur une machine parallèle

Résumé— Ce chapitre présente une méthode pour déterminer les temps d'exécution extrêmes (temps minimal et temps maximal) d'un programme sur une architecture donnée. Le programme est un code ordinaire représentant le comportement d'un circuit. L'architecture est caractérisée par l'interconnexion d'un ensemble d'unités fonctionnelles (e.g., additionneurs, multiplicateurs) et d'un ensemble d'unités de stockage (RAM, banques de registres). C'est une architecture *paramétrable* de type VLIW (very large instruction word).

Cette méthode a deux avantages principaux par rapport aux méthodes précédentes:

(1) C'est une méthode statique qui se base sur l'analyse de la structure du programme et non sur des simulations dynamiques du programme. Les méthodes basées sur la simulation sont lentes et leurs résultats sont sensibles aux données avec lesquelles le programme a été simulé.

(2) La méthode est applicable à des programmes qui contiennent aussi bien des instructions de traitement que des instructions de contrôle. Par contre, les méthodes statiques précédentes sont restreintes à des programmes qui ne contiennent que des instructions de traitement.

Le reste de ce chapitre est constitué d'une version étendue de l'article [BEN96b].

Performance Analysis for Hardware/Software Co-synthesis

Imed E. Bennour†, Michel Langevin†† and El M. Aboulhamid†
† *Département d'informatique et recherche opérationnelle, université de Montréal*
CP 6128, Centre ville, H3C-3J7,PQ, Canada
††*GMD-SET, Schloβ Birlinghoven, D-53757 Sankt Augustin, GERMANY*

Abstract— This paper presents a method for estimating the extreme case bounds (upper-bound and lower-bound) on the running time of a source program on a target hardware architecture. The source program may be any block of code containing data processing and control statements. The target architecture is specified by a set of functional units, a set of storage units and an interconnection network.

3.1. introduction

Hardware/Software (HW/SW) co-synthesis [CAM96] refers to the design approach which mixes hardware and software implementations of complex numerical systems in order to reduce the design cost while satisfying the performance requirements. A hardware implementation has better performance, whereas a software implementation has lower cost and allows later modifications. Figure 3.1 shows the general process of HW/SW co-synthesis. One of the key task in this process is the partitioning of the original system description into hardware and software modules. Hardware modules will be implemented as a dedicated hardware, while software modules will be implemented as programs running on predesigned general-purpose processor. The decision to map functionalities into software or hardware parts is based on estimates of achievable performance and the implementation cost of the respective parts. An automatic tool to estimate the performances of hardware and software implementations (tasks A and B in Figure 3.1) enables the designer to quickly evaluate different design alternatives.

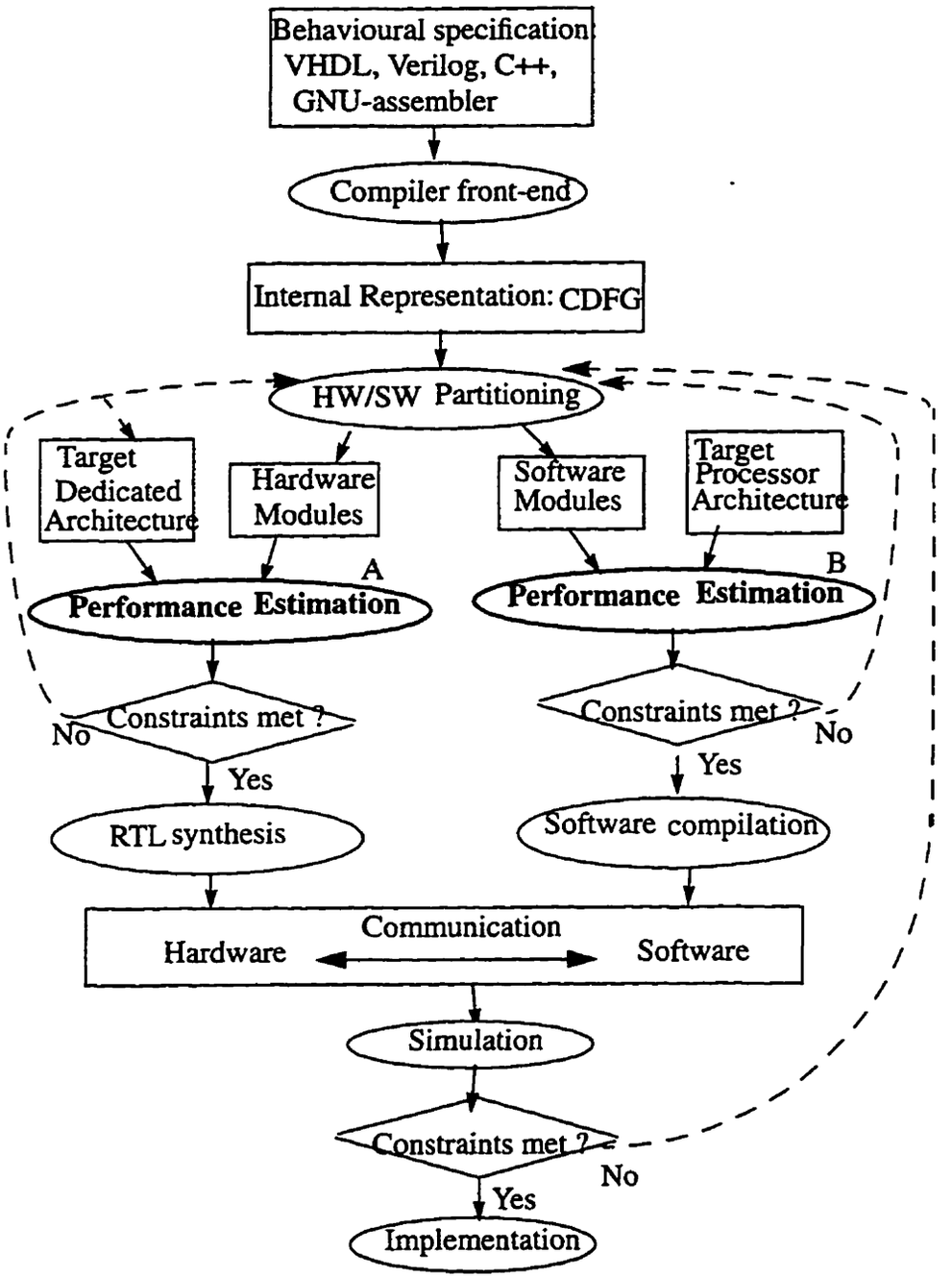


Figure 3.1 General framework of HW/SW co-synthesis.

This paper presents a method for estimating the upper-bound and the lower-bound on the performance of a control-data flow graph (CDFG) on a target hardware architecture. The target architecture is defined by a set of functional units (e.g., ALUs, multipliers), a set of storage units (RAMs, register-files), and an interconnection network. t_{LB} and t_{UB} are extreme case bounds on the performance of a CDFG, if for any input data the running time of the program belongs to $[t_{LB}, t_{UB}]$. Tight values of t_{LB} and t_{UB} not only give a good estimation of the achievable performance but they also permit to check if performance requirements can be satisfied by a target architecture. Let t_{max} denotes the maximal allowed delay of a CDFG. If $t_{max} \geq t_{UB}$, it means that for any input data the performance constraint will be satisfied. If $t_{max} < t_{LB}$, it means that the performance constraint cannot be met by the target architecture.

Several algorithms have been proposed in the literature [SHA93, LAN93, CHA94, RIM 94] for determining lower-bound and upper-bound on the performance of a given specification under resource constraints. However these algorithms are limited to data flow graphs without control statements, which is very restrictive.

The rest of this paper is organized as follows. Section 3.2 describes the estimation model used, and Section 3.3 presents the estimation method. Experimental results are given in Section 3.4.

3.2. Model of Estimation

There are two possible ways to estimate the extreme case bounds on the performance of a program on a target architecture: dynamic simulation and static estimation. Dynamic simulations consist of simulating the program execution with different input data, whereas static estimations are based on the analysis of the program structure. Static estimations are faster than dynamic simulations and insensitive to input data. The estimation approach presented in this work is static.

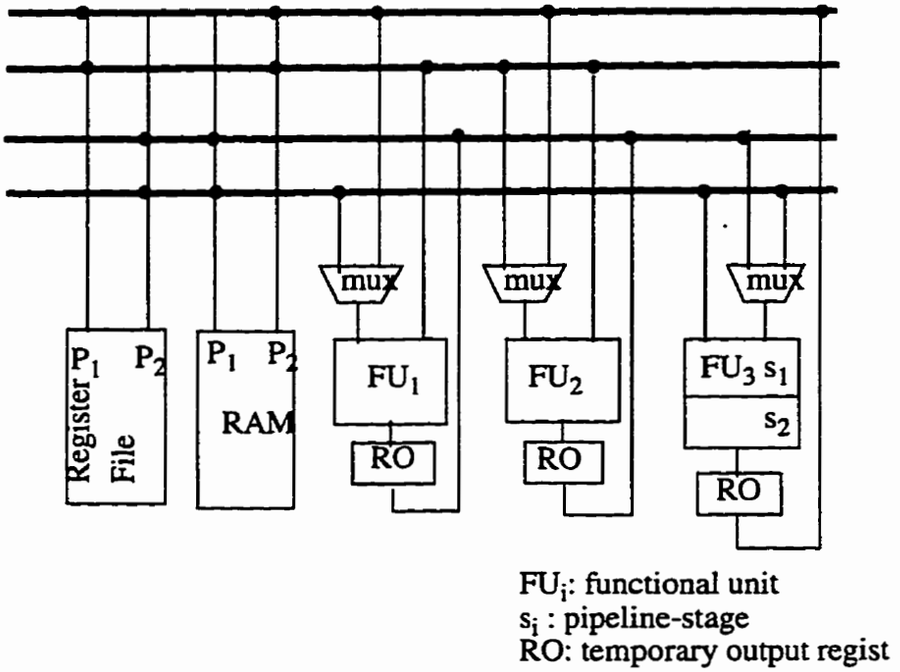


Figure 3.2 An instance of the parametrized bus-based architecture.

3.2.1. The hardware architecture model

Development of an estimation technique requires the definition of the target architecture model. In this work, we use a parametrized bus-based architecture as target model. Figure 3.2 shows an instance of the parametrized bus architecture. The parameters defines:

- the number of functional units of each type, their speed and their pipeline-stage number;
- the number of memory ports;
- the number of register-file ports;
- the number of buses;
- and the connections between components.

The choice of the parametrized bus architecture was influenced by the following reasons:

1) It enables the specification of a wide variety of target architectures, ranging from completely sequential to massively parallel architecture;

2) The bus architecture is frequently used in microprocessor systems;

3) Most high-level synthesis systems use bus-based architectures.

3.2.2. The control data-flow graph (CDFG) model

A CDFG [GAJ92] is a graphical representation of source programs. A CDFG is a composed of two type of graphs: the control flow graph (CFG) and the data-flow graph (DFG). A CFG is defined by a pair (BB, E), where BB is the set of nodes representing basic blocks, and E is the set of edges representing precedence execution order between basic blocks. To each basic block bb is associated a DFG (O_{bb}, A_{bb}), where O_{bb} is the set of nodes representing atomic operations such additions and multiplications, and A_{bb} is the set of arcs representing precedences between atomic operations. An execution of the CDFG consists of a sequential execution of basic blocks. Figure 3.4 shows the CDFG of the source code given in Figure 3.3. Feedback edges in the CFG represent loop statements.

We assume that each loop statement has a bounded number of iterations, otherwise the worst running time cannot be computed in general; it is well known that if a program contains unbounded loop statements then it is not possible to decide if its execution terminates. The maximal numbers of loop iterations are annotated in the source code by the user. The user can also specify the minimal numbers of loop iterations; default values are equal to zero. A *false-path* is a path in the CFG which is never executed due to incompatibility of two or more conditional branching. For example, if the conditions c_2 and c_3, in Figure 3.4, cannot be true at the same time, then the sequence of basic blocs $bb_d bb_f bb_g$ is never executed.

```
y := 2*3;
while (c1) {
   if (c2) then z := (x+5) * (5 +y) ;
   else bbe;
   if (c3) then bbg;
   else {
        while (c4) {
             bbj ; bbk; }
   }
   bbi ;
}
```

Figure 3.3 A source program example

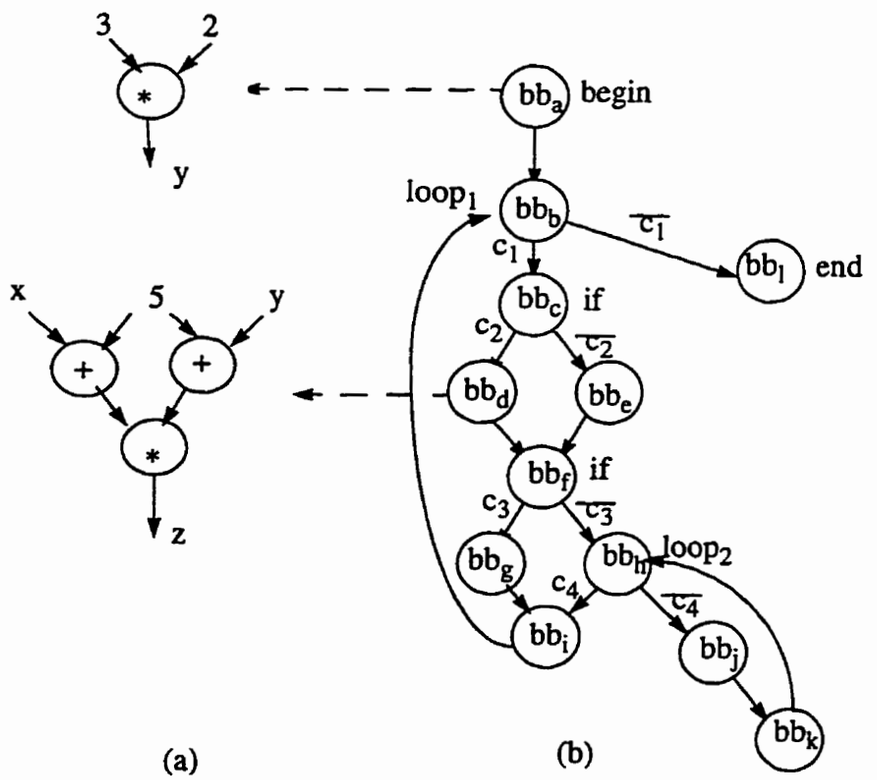


Figure 3.4 The CDFG of the program given in Figure 3.3: (a) DFG, (b) CFG.

3.3. Extreme case performance bounds

3.3.1. Upper-bound estimation

The proposed technique for determining an upper bound on the performance of a CDFG is composed of two steps: (1) determining an upper bound on the performance of each basic block, and (2) deducing an upper-bound for the whole CDFG.

3.3.1.1. Upper-bound on the performance of a basic block

The running time of a basic block is equal to the time necessary to execute the operations in the corresponding sub-DFG. An Upper-bound on the performance of a basic block can be obtained by scheduling the corresponding DFG on the target hardware architecture. As we use a parametrized bus-based architecture as a target model, we developed a list scheduling algorithm similar to the one in [GAJ94].

3.3.1.2. Upper-bound on the performance of a CDFG

We define the length of a path in a CFG as the sum of the running times of basic blocks forming this path. An upper-bound on the performance of a CFG will be the length of the longest path when considering that (1) the running times of basic blocks are equal to their upper-bound values, and (2) each loop is executed a maximal number of times.

One way of determining the length of the longest path in a CFG consists of first completely unfolding all loops and then computing the length of the longest one in the unfolded graph. The disadvantage of this method is that size of the unfolded graph will be very large if the loop-iteration numbers are high. The method that we use does not require loop unfolding. This method is described by the algorithm given in Figure 3.5.

Following is an illustration of this algorithm with the CFG of Figure 3.6a. The number beside a node is the performance upper-bound of the corresponding basic block, while the number beside a feedback edge is the maximal iteration number of the corresponding loop. $loop_2$ is the most inner loop, which is executed at most 20 times and each iteration takes (7+15+2) cycles. Thus, the running time of this loop will take at most 480 clock-

cycles. In Figure 3.6.b $loop_2$ is replaced by a single node with a weight equal to 480. After repeating the same process with $loop_1$ we obtain the graph of figure 3.6.c. We deduce that 5002 clock-cycles is an upper bound on the performance.

Notice that tighter upper-bound on the performance of a CDFG can be obtained if optimal DFG schedules are computed and/or if only CFG feasible-paths are considered. Unfortunately, each of these problems is NP-complete. An integer linear programming formulation was presented in [MAL95], where some false-paths of the CFG could be avoided during the estimation.

Input: a CDFG, an upper-bound on the performance of each basic block, a maximal number of iterations of each loop;
Output: upper-bound on the performance of the CDFG;

Begin

-Associate to each basic block in the CFG a weight equal to its performance upper bound;

Repeat:

For all inner-most loops **Do**:

- Compute the length (l) of the longest path in the current loop body;
- Replace in the CFG the loop by a single node;
- Associate to this node a weight equal to (l * the maximal number of iterations of the loop);

End For

Until no more loops in the CFG

- Compute the length of the longest path in the resultant graph. Return this length as an upper bound on the performance of the CDFG;

End

Figure 3.5 An algorithm for determining an upper-bound on the performance of a CDFG.

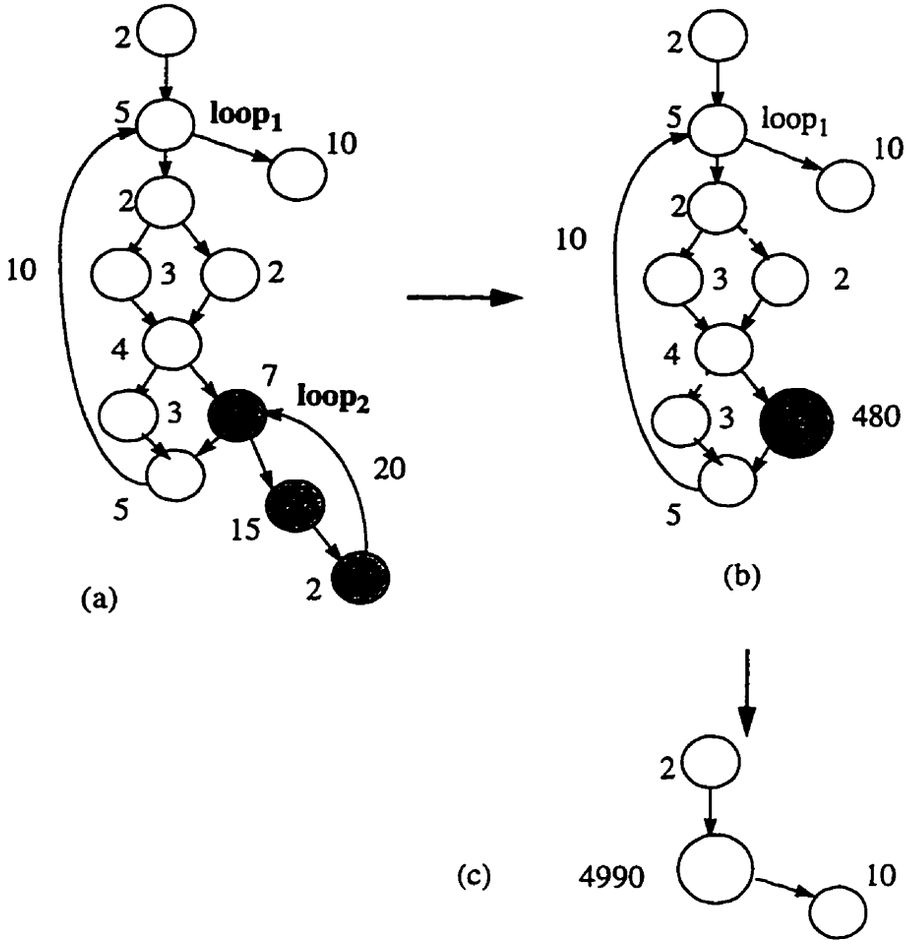


Figure 3.6 Illustration of the performance upper-bound algorithm.

3.3.2. Lower-bound estimation

A lower-bound on the running time of a CDFG can be derived from a lower-bound on the performance of each basic block and from a minimal number of iterations for each loop. The lower-bound will be the length of the shortest path in the CFG when considering (1) that the running times of basic blocks are equal to their lower-bound values, and (2) that each loop is executed a minimal number of times.

3.3.2.1. Lower-bound on the performance of a basic bloc

A trivial lower-bound on the performance of a basic block is the critical path delay of

its data flow graph (DFG) where the weights of nodes are set to operation-durations. Tighter lower-bounds can be obtained by taking into account resource constraints.

Notations.

- $G_{bb} = (O_{bb}, A_{bb})$ is the DFG of a basic-block *bb*. We assume that all multi-cyclic operations are broken into multiple uni-cycle operations.
- o_i is a node of G_{bb},
- $ASAP_{o_i}$ is the "as soon as possible" starting execution time of operation o_i,
- $ALAP_{o_i}$ is the "as late as possible" starting execution time of operation o_i,
- CP is the critical path delay of G_{bb},
- $T_r \subset O$ is the set of operations to be executed by functional units of type r,
- M_r is the number of functional units of type r available,

In [RIM94], Rim *et al.* have proposed an algorithm (Figure 3.7) for determining a lower-bound on the schedule length of a basic block under functional unit constraints. The computational complexity of this algorithm is $O(|O_{bb}| + CP \cdot LB)$ where LB is the returned lower-bound. We have reduced this complexity to $O(|O_{bb}| + CP^2)$. Notice that LB is always greater or equal to CP and in the worst case it can be equal to $|O_{bb}|$. The new algorithm is described in Figure 3.8.

The first step of this algorithm can be done in $O(|O_{bb}|)$ by traversing the data-flow graph. The second step can also be done in $O(|O_{bb}|)$ using the *bucket sort* [AHO83], since we know that the maximal value of $ALAP_{o_i}$ is equal to the critical path delay CP. For the third step, we used the following data structure [RIM94]. Each operation o_i has a pointer to a box containing the earliest time-step where it can be scheduled, which is initially equal to $ASAP_{o_i}$. Operations having the same $ASAP$ values point to the same box (Figure 3.9). Thus, the maximal number of boxes is equal to CP. Once an operation is assigned to a time-step, the value of its box is updated to this time step. As each box can be updated at most CP times, the maximal number of updating is at most CP^2. Finally, step 4 is done in $O(CP)$. Thus, the computational complexity of our algorithm is $O(|O_{bb}| + CP^2)$.

---

input: DFG of a basic block, set of resources $\{M_r\}$,
output: lower bound (*LB*) on the schedule performance,
begin

1. Find the $ASAP_{o_i}$ and $ALAP_{o_i}$ values of all operations;
2. Sort the set $\{o_i\}$ in the increasing order of $ALAP_{o_i}$ values;
3. **For** each operation o_i **do**

   Assign it to the earliest time-step satisfying $ASAP_{o_i}$ value and resource constraints but ignoring precedence constraints. Let S_{o_i} this time-step.

4. $LB \leftarrow \max_{o_i} \{S_{o_i} - ALAP_{o_i}\} + CP;$

end

Figure 3.7 Rim's algorithm for computing a lower-bound [RIM94].

---

input: DFG of a basic block, set of resources $\{M_r\}$,
output: lower bound (*LB*) on the schedule performance,
begin

1. Find the $ASAP_{o_i}$ and $ALAP_{o_i}$ values of all operations;
2. Sort the set $\{o_i\}$ in the increasing order of $ALAP_{o_i}$ values;
3. **For** each operation o_i **do**

   If it is possible assigned it to the earliest time-step between $ASAP_{o_i}$ and $ALAP_{o_i}$ satisfying resource constraints but ignoring precedence constraints. Otherwise assigned it to the time-step equals to $ALAP_{o_i}$, even if resource constraints are not satisfied.

   $LB \leftarrow 0;$
4. **For** j = 1 to CP **do**

   $LB \leftarrow \max_r (LB, \left\lfloor \frac{n_j}{M_r} \right\rfloor + CP)$, where n_j is the number of operations of a type r assigned to time-step j;

end

Figure 3.8 An improved version of Rim's algorithm

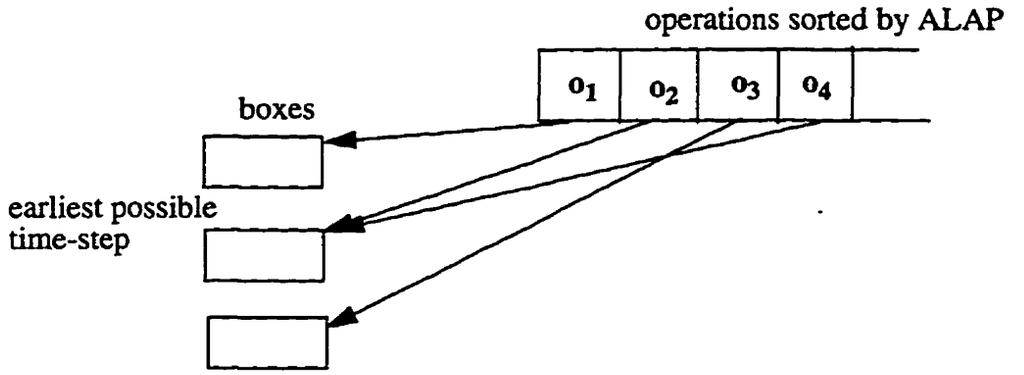


Figure 3.9 A data structure for assigning operations.

Input: a CDFG, an lower-bound on the performance of each basic block, a minimal number of iterations of each loop;
Output: lower-bound on the performance of the CDFG;

Begin

-Associate to each basic block in the CFG a weight equal to its performance lower-bound;

Repeat:

For all inner-most loops **Do**:

- Compute the length (l) of the shortest path in the current loop body;
- Replace in the CFG the loop by a single node;
- Associate to this node a weight equal to (l * the minimal number of iterations of the loop);

End For

Until no more loops in the CFG

- Compute the length of the shortest path in the resultant graph. Return this length as an lower-bound on the performance of the CDFG;

End

Figure 3.10 An algorithm for determining a lower-bound on the performance of a CDFG.

3.3.2.2. Lower-bound on the performance of a CDFG

The algorithm for computing the length of the shortest path is similar to the one used for computing the length of the longest path, except that for each loop we consider its shortest running time instead of longest running time. This algorithm is described in Figure 3.10.

3.4. Experimental results and conclusions

In Tables 3.1 and 3.2, we report part of the experimental results obtained with two benchmarks: square matrix multiplication and matrix convolution. The first and the second benchmark contain three and two nested loops, respectively. In both benchmarks, nested loops have the same iteration number which is equal to matrix dimension. The source codes of these benchmarks were translated in GNU-assembler before analysis. The running time of the tool was in the order of seconds. The first column in Tables 3.1 and 3.2 indicates the iteration bounds of loops, and the last two column indicate the extreme case bounds on the performance under the resource constraints. From Table 3.1, we deduce that the performance of the first benchmark decreases by using more ALUs and/ or register file ports. However, more than 3 ALU and/or 3 register file ports does not reduce the worst performance any more. We can deduce also that RAM ports are not critical resources. Results in Table 3.2 shows that all resource types are critical, and more than 4 ALU and/or 3 memory ports and/or 2 register file ports does not reduce the worst performance any more.

The estimation tool was developed in C++ using the co-synthesis SIR/CASTLE (Codesign and Architecture-driven Synthesis TooL Environment) database [CAM96].

Acknowledgements. This work was partially sponsored by the FCAR (Fonds pour la Formation de Chercheurs et l'Aide à la Recherche) from Quebec gouvernement.

Table 3.1: Matrix multiplication

Matrix size	# of ALU	# of RAM ports	# of regis. file ports	Perform. lower bound (clock cycles)	Perform. upper bound (clock cycles)
50×50	1	1	1	$1.015\ 10^6$	$1.283\ 10^6$
	1	1	2	$1.015\ 10^6$	$1.150\ 10^6$
	1	1	3	$1.015\ 10^6$	$1.147\ 10^6$
	1	2	1	$1.015\ 10^6$	$1.283\ 10^6$
	1	2	1	$1.015\ 10^6$	$1.283\ 10^6$
	2	1	1	$0.510\ 10^6$	$1.280\ 10^6$
	2	1	2	$0.510\ 10^6$	$0.770\ 10^6$
	2	1	3	$0.510\ 10^6$	$0.767\ 10^6$
	2	2	1	$0.510\ 10^6$	$1.280\ 10^6$
	3	1	3	$0.385\ 10^6$	$0.642\ 10^6$
	3	1	4	$0.385\ 10^6$	$0.642\ 10^6$
	4	1	3	$0.385\ 10^6$	$0.642\ 10^6$
500×500	1	1	1	$1.001\ 10^9$	$1.250\ 10^9$
	1	1	2	$1.001\ 10^9$	$1.121\ 10^9$
	2	1	1	$5.011\ 10^8$	$1.253\ 10^9$
	2	1	2	$5.011\ 10^8$	$7.521\ 10^8$
	3	1	1	$3.760\ 10^8$	$1.253\ 10^9$
	3	1	2	$3.760\ 10^8$	$7.520\ 10^8$

Table 3.2: Matrix convolution

Matrix size	# of ALUs	# of RAM Ports	# of RF Ports	Lower-bound (clock cycles)	Upper Bound (clock cycles)
50X50	1	1	1	$7.941\ 10^4$	$8.211\ 10^4$
	1	1	2	$7.941\ 10^4$	$8.206\ 10^4$
	2	1	1	$4.111\ 10^4$	$5.141\ 10^4$
	2	1	2	$4.111\ 10^4$	$4.381\ 10^4$
	2	2	1	$4.100\ 10^4$	$5.136\ 10^4$
	2	2	2	$4.100\ 10^4$	$4.371\ 10^4$
	3	2	1	$3.080\ 10^4$	$5.126\ 10^4$
	3	3	1	$2.820\ 10^4$	$5.126\ 10^4$
	3	3	2	$2.820\ 10^4$	$3.095\ 10^4$
	3	3	3	$2.820\ 10^4$	$3.095\ 10^4$
	4	3	2	$2.820\ 10^4$	$2.835\ 10^4$
	5	3	2	$2.820\ 10^4$	$2.835\ 10^4$
500X500	1	1	1	$7.769\ 10^6$	$8.021\ 10^6$
	2	1	1	$4.011\ 10^6$	$5.014\ 10^6$
	1	2	1	$4.011\ 10^6$	$8.211\ 10^4$
	2	1	1	$4.111\ 10^4$	$5.141\ 10^4$

Chapitre 4.

L'Allocation de registres dans la synthèse de haut-niveau: nouvelle approche.

Résumé— Cet chapitre traite le problème d'allocation des registres aux variables dans la synthèse de haut-niveau des circuits dédiés à des applications spécifiques. Nous présentons une nouvelle organisation de registres dans le chemin de données sous formes de files circulaires. Cette organisation permet de réduire le nombre de registres et le nombre de signaux de contrôle dans le circuit. Elle constitue une alternative aux banques de registres et une extension des files simples introduites dans [ALO94].

Le problème d'allocation de registres que nous traitons est le suivant: Etant donné un ensemble de variables définies par leur durées de vie, trouver une assignation des variables aux files circulaires qui minimise le nombre de files et le nombre de registres dans ces files.

Ce problème a été modélisé et résolu efficacement en utilisant le paradigme de la programmation à contraintes logiques et à arithmétique des intervalles.

Le reste de ce chapitre est constitué d'une version étendue de l'article [BEN96c].

Register Allocation Using Circular FIFOs

Imed Eddine Bennour et El Mostapha Aboulhamid
Département d'informatique et recherche opérationnelle, université de Montréal
CP 6128, Centre ville, H3C-3J7,PQ, Canada

Abstract—In this paper, we study the memory allocation problem in data-path synthesis. We propose a new register organization called circular FIFO as an alternative to register file organization and as an extension of queues. In comparison with register file organization, FIFO organization eliminates the overhead of address generation, decoding hardware, and the extra access delay. The memory allocation problem, based on the circular FIFO organization, has been solved efficiently using constraint logic and interval constraint programming. Case studies, much more complex than the existing high level synthesis benchmarks, have been solved in less than 3 minutes.

4.1. Introduction

A common approach to behavioral synthesis involves data-flow graph scheduling, functional unit allocation, interconnection and memory allocation. Memory allocation maps constants and variables of a data-flow graph to storage elements (e.g., ROM, registers, register files). Both storage elements and their control part occupy a significant portion of the chip area. Therefore, it is important to minimize the number of storage elements and to organize them in such a way that control, address generation and decoding hardware are reduced. This is the goal of the work presented in this paper.

Conventional memory allocation approaches [DEM94, GAJ92, MIC92] can be classified into two categories. In the first category, variables are mapped to registers based on a lifetime analysis of variables. The lifetime interval of a variable is the time interval between its first value assignment and its last use. Multiple variables can share the same

register if their lifetime intervals do not overlap with one another. Lifetime analysis approaches aim to minimize the number of registers but not the control hardware. These approaches are effective only when the number of registers is small, otherwise the number of interconnections and multiplexers become very large and the hardware architecture becomes very irregular. In the second category, variables are mapped to registers which are then grouped into register files (or multiport memories) based on disjoint access time. Registers can be grouped in the same register file if they are not accessed simultaneously. Register file organization is profitable only if the number and the size of register files are small: a large number on register files not only adds more address generation and decoding hardware, but also leads to longer access delay due to the decoding circuits and long data driving lines. Recently, Aloqeely [ALO94] has proposed the use of *sequencers* as an alternative to register files. Queue and stack are examples of sequencers. However, his approach has some limitations: first, except queues, sequencers are costly in implementation and, therefore, they cannot be used in a large number; second, the allocation method is suitable only for regular iterative applications where the access patterns of variables are highly regular and uniform, e.g., most variables have the same lifetime duration.

In this paper, we propose a new register organization, called circular FIFO, as an alternative to separated register organization and register file organization, and as an extension of queues [ALO94]. *A circular FIFO* is a row of shift registers where the output of a register is connected to the input of the following one, and the output of the rightmost register is optionally connected to the input of the leftmost register. A circular FIFO is shown in Figure 4.1 The leftmost register (R_0) and the rightmost register (R_{M-1}) are called *the head* and *the tail of the FIFO*, respectively. Data enters the FIFO from the head and is visible only at the tail. When a data item is inserted in the FIFO head all registers are shifted. The circularity in the FIFO adds more flexibility to data access capability: it allows multi-access to the same data, and it removes the first-in first-out constraint on data, since a data item can be reinjected in the head once it reaches the tail. Separated register organization is a special case of the FIFO organization where each FIFO contains

only one register.

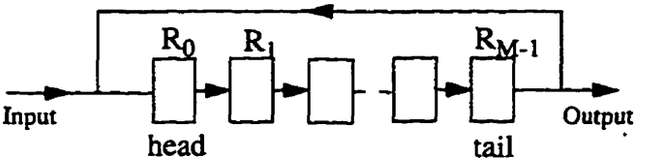


Figure 4.1 Circular FIFO.

By grouping registers into FIFOs, we reduce the number of control signals. Moreover, the use of FIFO structures may reduce the number of storage registers, since variables are not constrained to be kept in the same registers during all their lifetime. Comparing with file registers, circular FIFOs do not need memory address decoding hardware, and hence they do not suffer from decoding delays, which grow proportionally with the size of a register file. Figure 4.2 shows the register file based architecture and the FIFO based architecture. In the FIFO based architecture register files may still be used, but in a reduced number.

To resolve the FIFO allocation problem, we used CLP-BNR [BNR93], a constraint logic programming (CLP) language, based on prolog augmented with relational interval arithmetic. CLP-BNR provides a unified framework to express and solve dynamically a set of constraints over reals, integers and booleans using interval narrowing techniques.

The organization of the paper is as follows. Section 4.2 defines the FIFO allocation problem. Section 4.3 presents an overview of the interval constraint paradigm, then a formulation of the FIFO allocation problem using interval constraints. Section 4.4 describes how handle the case of iterative behaviors. Experimental results are presented in section 4.5.

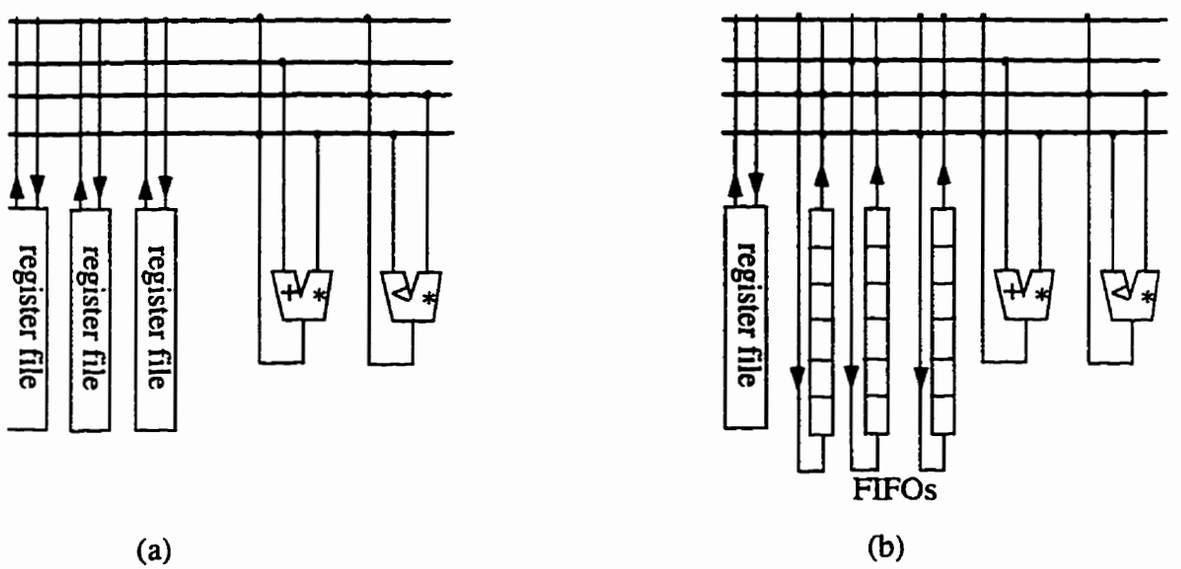


Figure 4.2 (a) Register file based architecture (b) FIFO based architecture

4.2. Problem definition

A circular FIFO is defined by a set of shift registers $(R_0, R_1, \ldots, R_{M-1})$, where R_i is connected to R_{i+1}, for $i = 0, \ldots, M-2$, and R_{M-1} is connected to R_0. A Circular FIFO is controlled by three operations:

- *Insert* (v_i): insertion of a variable labelled v_i in the FIFO header. After an insertion all registers of the FIFO are right shifted and the data item in the FIFO tail is lost.
- *Shift*: right shift all registers.
- *Rotate*: it is equivalent to *Insert (content of the tail).*

During a control-step, at most one control-operation can be performed on the FIFO.

A variable v_i is defined by its write-time and its read-time(s), fixed by the scheduling task.

$$v_i: (\textit{Write-time}_i, \textit{Read-time}_i^{1}, \textit{Read-time}_i^{2}, \ldots, \textit{Read-time}_i^{n_i}).$$

v_i should be inserted in a FIFO at $\textit{Write-time}_i$, and it should be available in the FIFO tail at each $\textit{Read-time}_i^{j}$, for $j = 1, \ldots, n_i$. The lifetime interval of v_i is equal to $\left[\textit{Write-time}_i, \textit{Read-time}_i^{n_i}\right]$.

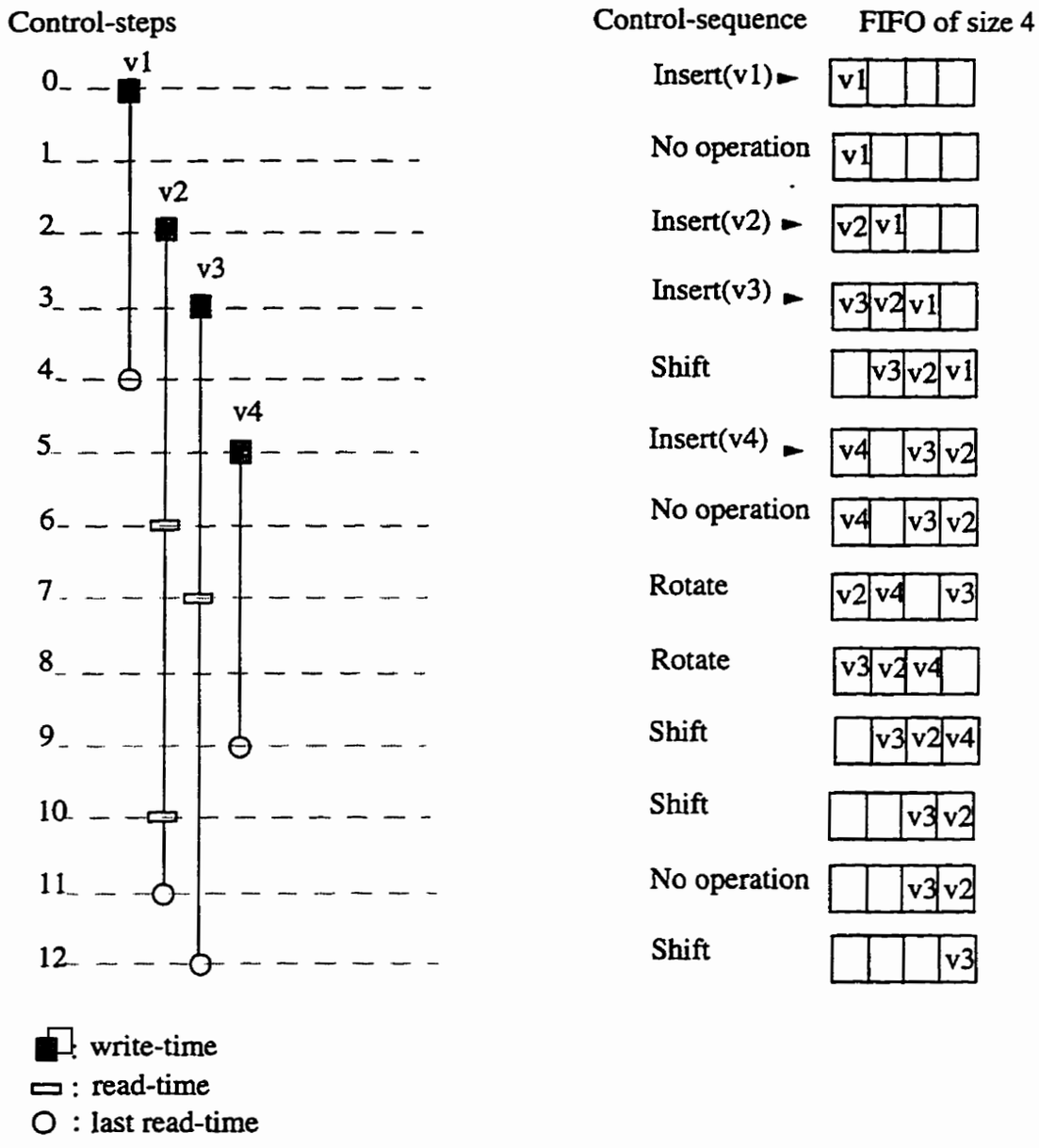


Figure 4.3 Illustration of the functioning of the circular FIFO

The example in Figure 4.3 illustrates the functioning of the circular FIFO. We observe that the circularity in the FIFO allows multi-access to a same variable (e.g., v_2), and it removes the first-in first-out constraint (e.g., v_3 enters the FIFO before v_4 and leaves after

v_4). Notice that the FIFO size should always be greater than the maximal number of variables alive at the same time, and smaller than $\underset{v_i}{Min}\left(Read\text{-}time_i^{1} - Write\text{-}time_i + 1\right)$, otherwise some variables will not have enough time to traverse the FIFO from the header to the tail.

A *control sequence* of a circular FIFO is a sequence of control operations (insert, rotate, shift, no operation). A control sequence is *valid* for a set of variables, if it guarantees that each variable in the set is inserted in the FIFO at its writing time, and it is available in the tail at its reading time(s). It may happen that, no valid control sequence exists for a set of variables, i.e., variables in this set cannot be mapped into a same FIFO.

Now we can define the two problems related to allocating variables to circular FIFOs.

- **The single-FIFO allocation problem:** Given a set of variables $V = \{v_i\}$, defined by their write-time and read-times, can these variables be mapped to a same circular FIFO? If they can, what is the minimum size of such a FIFO?
- **The multi-FIFO allocation problem:** Given a set of variables $V = \{v_i\}$, defined by their write-time and read-time(s), find a mapping from V to circular FIFOs that optimizes the number of FIFOs and the total number of registers.

We suspect that the single-FIFO allocation problem is NP-complete. We solve it exactly using CLP and interval constraint paradigms. The multi-FIFO allocation problem is NP-hard. A heuristic approach is taken to resolve it. It is based on iterative resolutions of the single-FIFO allocation problem. The following steps summarize this heuristic (developed also using the same CLP environment):

– The set of variables V is divided into disjoint *clusters* (subsets) C_l such that: (1) variables in the same cluster do not have either the same writing-times nor the same reading-times, (2) if two variables v_i and v_j belong to a same cluster and the writing-time of v_j is greater than the writing-time of v_i then the reading-time of v_j is greater than the reading-time of v_i, (3) in each cluster C_l, the maximal number of variables in C_l alive at the same time is smaller than or equal to the minimal live-time among all variables in C_l. These criteria increase the likelihood of mapping successfully all variables in a cluster to the same circular FIFO.

– For each cluster, check if it can be mapped to the same circular FIFO using the exact resolution of the single-FIFO problem. If not, the variables causing the failure are removed from the current cluster and are redistributed on other clusters if possible. New clusters are added if necessary. This process is repeated until all the clusters are mapped to some feasible circular FIFOs.

– The last step is to reduce the number of clusters, i.e., the number of circular FIFOs. Repetitively, we pick the cluster containing the minimal number of variables, then we try to redistribute all its variables on the other clusters. We check if a variable can be added into a cluster by solving the single-FIFO problem. This process is repeated until all the clusters are considered.

4.3. Formulation of the single-fifo allocation problem using interval constraints

4.3.1. A brief overview of the interval constraint paradigm [MOO79, BNR93]

CLP-BNR is a constraint logic programming system based on relation interval arithmetic. The use of interval arithmetic allows reasoning about domains of variables rather than fixed values. An interval is a closed bounded set of numbers, it defines either a continuous range of real numbers laying between a lower and an upper bound or a discrete range of integer values laying between integer bounds. The two endpoints of an interval X are denoted by $\overline{X}$ and $\underline{X}$. Thus, $X = [\underline{X}, \overline{X}]$. Two intervals are equal if their corresponding endpoints are equal.

Operations on intervals can be either the basic arithmetic operations defined on the reals (+, -, *, /, min, max, sin, cos, etc.), or arithmetic relations (equality, inequality, inclusion, etc.). In the following, we give the semantic of some of these operations.

• Arithmetic operations

$$[\underline{X}, \overline{X}] + [\underline{Y}, \overline{Y}] = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}]$$

$$-X = -[\underline{X}, \overline{X}] = [-\overline{X}, -\underline{X}] = \{-x \mid x \in X\}$$

$$max(X, Y) = [max(\underline{X}, \underline{Y}), max(\overline{X}, \overline{Y})]$$

- Arithmetic relations

  The *equality constraint* between two intervals X and Y, denoted $X == Y$, is *true* if the X and Y can be constrained to be equal by *narrowing* (reducing) X and/or Y.

  For example, if $X = [2, 5]$ and $Y = [4, 8]$, then $X == Y$ is true since both X and Y can be reduced to the same interval $[4, 5]$. If $X = [2, 5]$ and $Y = [6, 8]$, then $X == Y$ is false.

  The *less than or equal constraint* between an interval X and an interval Y, denoted by $X \leq Y$, is true if the interval X can be constrained to an interval Z where each element of Z is less than or equal to an element of Y.

  For example, if $X = [2, 15]$ and $Y = [6, 8]$, then $X \leq Y$ is true because X can be reduced to the interval $[2, 8]$ which is less than or equal the interval Y. If $X = [9, 15]$ and $Y = [6, 8]$, then $X \leq Y$ is false.

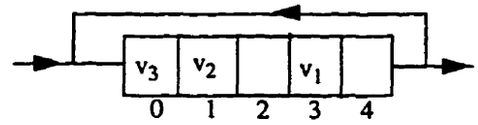
4.3.2. Formulation of the single-FIFO allocation problem

The formulation of the single-FIFO allocation problem is based on the successive states of the FIFO during the execution of a control sequence (sequence of shift, insert and rotate operation). If we label the shift registers composing a circular FIFO from 0 to M-1, then a FIFO state can be defined by the set of variables inside the FIFO and their position. The *distance* from a variable v_i to a variable v_j, denote by $Dist(v_i, v_j)$, is defined as following:

$$Dist(v_i, v_j) = (pos(v_j) - pos(v_i)) \ mod M$$

where $pos(v_i)$ is the v_i position inside the FIFO.

Example:



$$Dist(v_2, v_1) = (3 - 1) \ mod 5 = 2, \ Dist(v_2, v_3) = (0 - 1) \ mod 5 = 4$$

Based on the distance definition, we have:

$$Dist(v_i, v_j) \in [0, M-1], \forall v_i, v_j, \text{ where } M \text{ is the FIFO size}$$

$$Dist(v_i, v_i) = 0, \forall v_i$$

$$Dist(v_i, v_j) = M - Dist(v_j, v_i), \quad \forall v_i \neq v_j$$

$$Dist(v_i, v_j) = (Dist(v_i, v_k) + Dist(v_k, v_j)) \; modulo \; M, \quad \forall v_i \neq v_j \neq v_k$$

Notice that, due to the last equality, the distance is still defined even between variables which are not in the FIFO at the same time (i.e., variables which are not alive at the same time). Thus, the distance values between pairs of variables are sufficient to capture all the successive states taken by FIFO during the execution of a control sequence. It can be proven that finding a valid control sequence for a set of variables is equivalent to finding feasible distance values among variables.

To state the constraints which a distance between two variables should satisfy, we distinguish four cases depending on the precedence order between their write and read operations. These cases are shown in Figure 4.4.

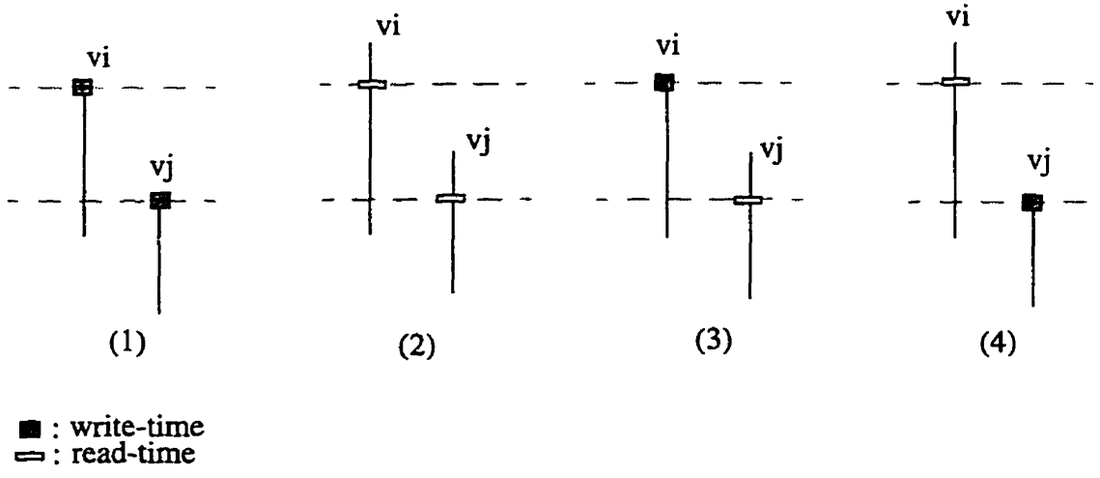


Figure 4.4 The four precedence orders between write and read operations

Case 1: *two successive write operations*

$Dist\,(v_j, v_i)$ should satisfy:

$$Dist\,(v_j, v_i) \leq Write\text{-}time_j - Write\text{-}time_i$$

because between the insertion-time of v_i and the insertion-time of v_j into the FIFO the maximal number of control operations (shift, insert, rotate) that can be done is equal to $(Write\text{-}time_j - Write\text{-}time_i - 1)$.

Case 2: *two successive read operations*

$Dist\,(v_j, v_i)$ should satisfy:

$$Dist\,(v_j, v_i) \leq Read\text{-}time_j^{\,k} - Read\text{-}time_i^{\,l}$$

This constraint guarantees that, after the reading-time of v_i there are still enough control-steps to propagate v_j until the FIFO tail.

Case 3: *write operation followed by read operation*

$Dist\,(v_j, v_i)$ should satisfy:

$$Dist\,(v_j, v_i) \leq Read\text{-}time_j^{\,k} - Write\text{-}time_i + 1$$

This constraint guarantees that, after the writing-time of v_i there are still enough control-steps to propagate v_j until the FIFO tail.

Case 4: *read operation followed by write operation*

$Dist\,(v_j, v_i)$ should satisfy:

$$Dist\,(v_j, v_i) = M - 1, \text{ if } Write\text{-}time_j = Read\text{-}time_i^{\,l}$$

$$Dist\,(v_j, v_i) \leq Write\text{-}time_j - Read\text{-}time_i^{\,l} - 1, \text{ otherwise}$$

If $Write\text{-}time_j = Read\text{-}time_i^{\,l}$, then the constraint states that when v_j is inserted in the FIFO header v_i should be in the tail; otherwise it states that between the reading-time of v_i and the writing-time of v_j in the FIFO the maximum number of shift and rotate operations that can be done is equal to $\left(Write\text{-}time_i - Read\text{-}time_i^{\,l} - 1 \right)$.

Figure 4.5 shows the set of distance constraints for the example used in Figure 4.3.

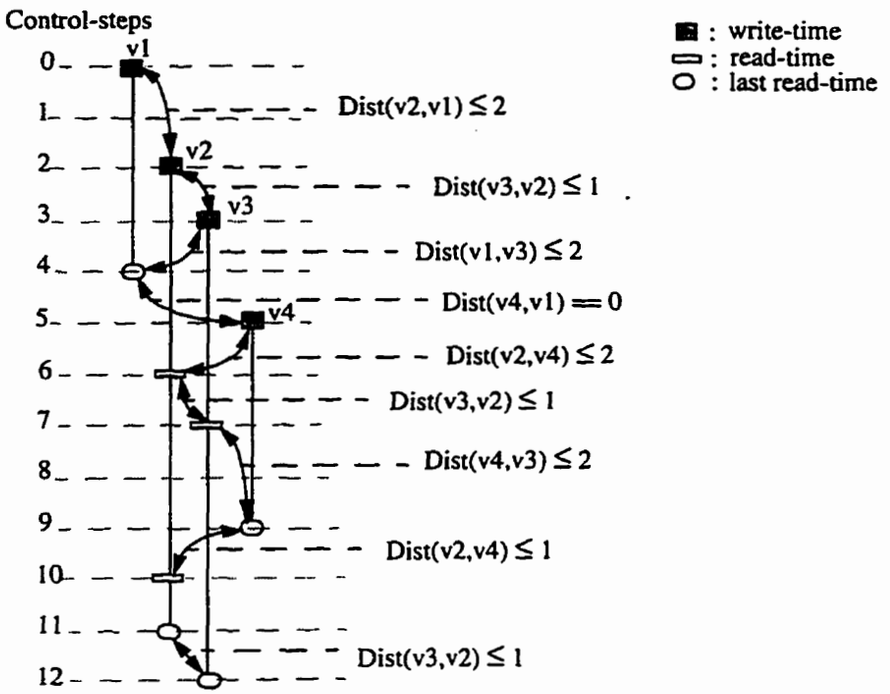


Figure 4.5 The set of distance constraints for the example used in Figure 4.3

Now we give the complete formulation of the single-FIFO allocation problem using the relational interval arithmetic:

Minimize the FIFO size M under:

(a) $\quad M \in [M\_min, M\_max]$

where M_min is the maximum number of variables alive at the same time, and

$$M\_max = \underset{v_i}{Min} \left(Read\text{-}time_i^{\,1} - Write\text{-}time_i + 1 \right)$$

(b) $\quad Dist(v_i, v_j) \in [1, M-1]$, $\forall v_i \neq v_j$ and v_i, v_j are alive at the same time

(c) $\quad Dist(v_i, v_j) \in [0, M-1]$, $\forall v_i \neq v_j$ and v_i, v_j are not alive at the same time

(d) $\quad Dist(v_i, v_j) == (M - Dist(v_j, v_i))$, $\forall v_i \neq v_j$

(e) $\quad Dist(v_i, v_j) == (Dist(v_i, v_k) + Dist(v_k, v_j))\ modulo\ M, \quad \forall v_i \neq v_j \neq v_k$

(f) $\quad Dist(v_i, v_j) \leq c_f$

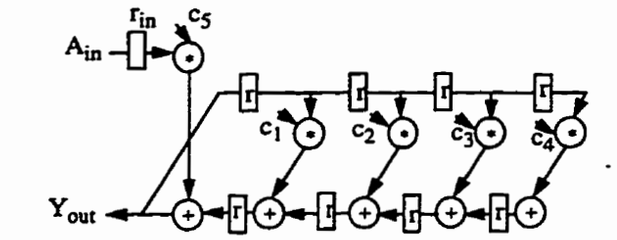
Constraint (a) defines the possible values of the FIFO size. Constraint (b) states that variables which are alive at the same time cannot share the same registers into the FIFO: their distances should be greater than zero. Constraints (c), (d) and (e) express the distance's properties. (f) is the set of distance constraints between variables that should be satisfied, as discussed previously in cases (1) to (4). It is important to mention that all these constraints are directly expressible in CLP-BNR.

4.4. Case of iterative behaviors

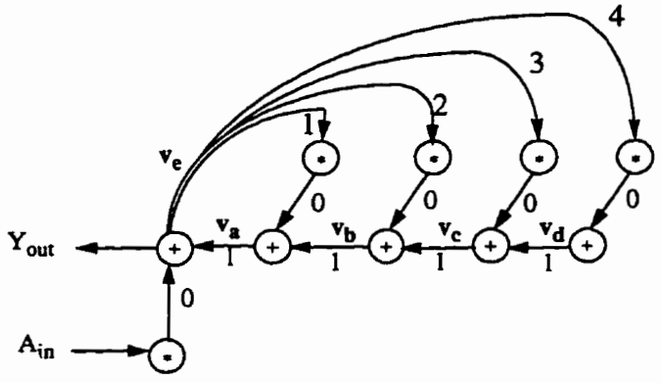
Now we consider variables which appear in iterative behaviors like infinite loop. We will use an example to explain how to handle this case.

Figures 4.6.a and 4.6.b show, respectively, the behavior of the polynomial divider benchmark and its cyclic precedence graph. The integer edge weight $H(o_i \rightarrow o_j)$ means that the data produced by operation o_i in any iteration k of the loop will be used by operation o_j in iteration $(k + H(o_i \rightarrow o_j))$. We assume that only one adder and one multiplier are available. Figure 4.6.c shows the variables' lifetimes obtained after scheduling the operations of the cyclic precedence graph. A second representation of variables' lifetimes, called circular representation, is shown in Figure 4.7.a. The perimeter of the circle is equal to the initiation interval value which is in this example equal to five. Variables that have lifetimes greater than the initiation interval should be split. For example, the variable v_d has a lifetime equal to six, it is split into v_d^1 and v_d^2 with lifetimes, respectively, of five and one. Now based on the circular representation, the distance constraints between variables' positions inside the FIFO can be specified as in the previous section (cases 1 to 4).

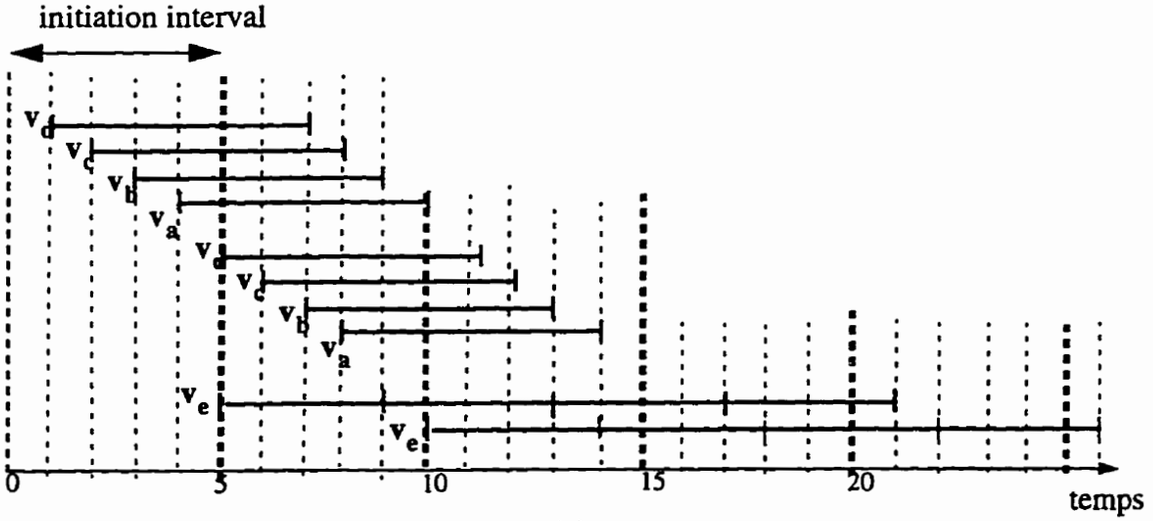
Figure 4.7.b shows the final implementation of the polynomial divider benchmark using the FIFO structure.



(a)



(b)



(c)

Figure 4.6 A polynomial divider (a) Its behavior (b) The cyclic precedence graph (c) Variables' lifetimes.

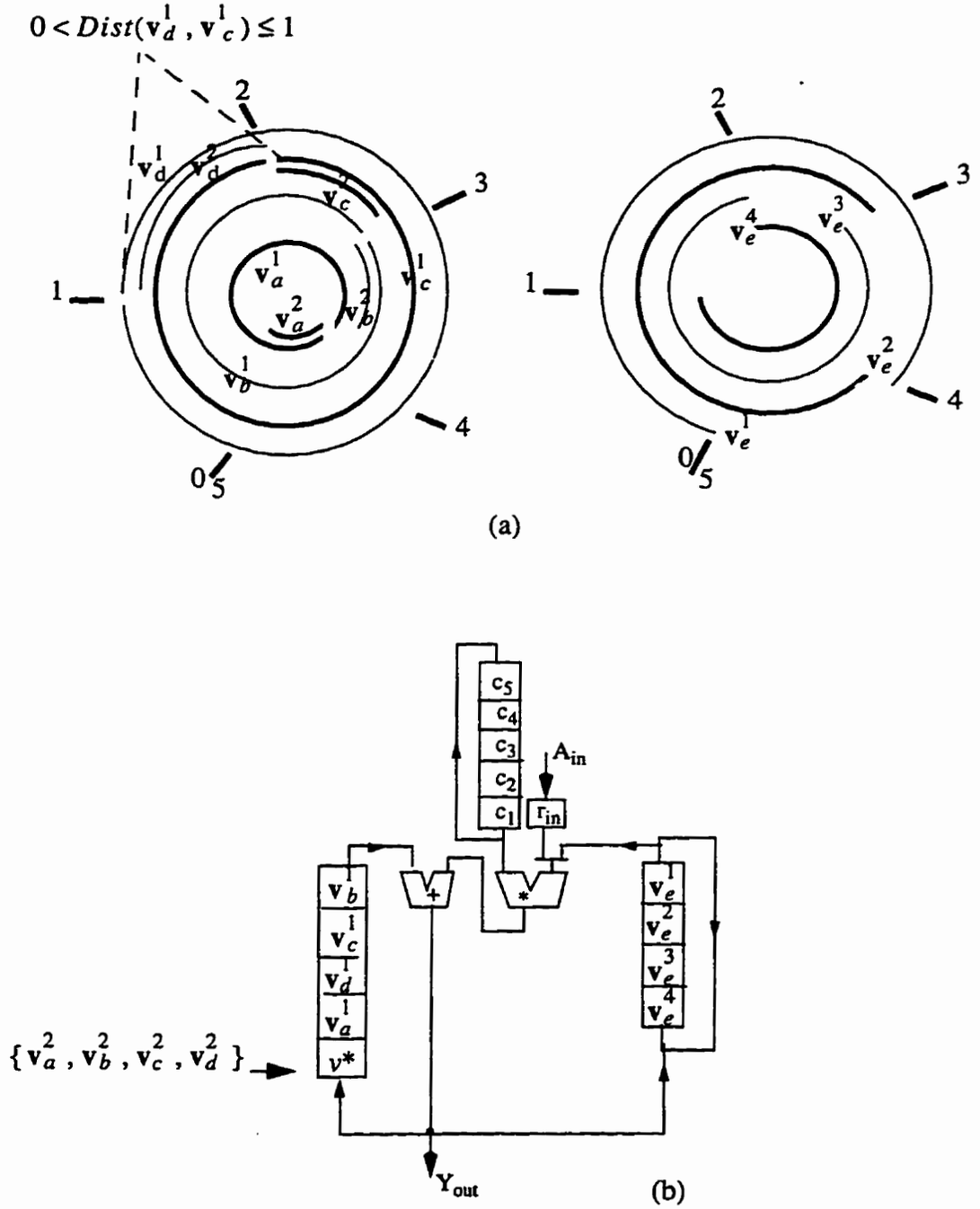


Figure 4.7 (a) Circular representation of variables' lifetimes (b) An implementation of the polynomial divider.

4.5. Implementation and experimental results

We present in Tables 4.1 and 4.2 experimental results obtained using the proposed circular FIFO allocation approach. Table 4.1 summarizes the results of four benchmarks frequently used by the high-level synthesis community. All benchmarks are scheduled using one adder and one multiplier. The second column of Table 4.1 indicates the minimal number of registers necessary to hold variables and constants. This number corresponds to the maximal number of variables and constants alive at the same time. The third column corresponds to the number of circular FIFOs obtained using our allocation approach. The total number of registers used in the final design is indicated in the fourth column. We observe that the number of registers used is always optimal, and that these registers are grouped in a small number of FIFOs, which reduces considerably the control signals. To evaluate the efficiency of our approach in general case, we have used random examples, which are more complex than usual high level synthesis benchmarks. The results are given in Table 4.2. For each example, we generate a set of variables with random lifetime intervals, then we perform the FIFO allocation algorithm. Our results are quite interesting. First, the number of circular FIFO (third column) is very small compared to the minimal number of registers (second column), in average each FIFO contains five to six registers. Second, the total number of registers used (fourth column) exceeds the optimum by 3 registers at most. We have performed the Aloqeely's allocation method [ALO94] on the same random examples, the number of queues returned is on average equal to the number of registers, i.e., most queues contain only one register and hence there is no gain compared to the separated register organization. This result is not surprising, since Aloqeely's approach assumes that most variables have the same lifetime durations, which is not the case for the randomly generated examples.

The low CPU times taken to solve the relatively large random examples (up to 100 variables) reflect the solving power of the interval constraint paradigm. By expressing the constraints of the problem over interval-valued variables rather then over integer variables, we have noted that the interval narrowing resolution technique reduces considerably the resolution time by identifying and removing invalid solution spaces.

Table 4.1: Benchmark results

Benchmark	Minimal # of registers	# of FIFOs	# of registers used	CPU time (sec.)
IIR filter	20	5	20	10
FIR filter	15	3	15	7
Polynomial divider	15	3	15	7
Three order filter	7	2	7	2

Table 4.2: Experimental results using random examples

# of variables	Minimal # of registers	# of FIFOs	# of registers used	CPU time (sec.)
20	14	3	15	10
20	15	4	17	16
20	12	4	12	7
20	11	3	12	18
20	11	3	13	9
30	18	3	19	171
30	16	3	18	44
30	18	4	18	32
30	17	3	17	494
30	20	4	21	63
50	26	4	27	85
50	23	5	26	86
50	22	4	25	76

Table 4.2: Experimental results using random examples

# of variables	Minimal # of registers	# of FIFOs	# of registers used	CPU time (sec.)
50	28	4	30	66
50	22	3	22	177
70	37	5	39	155
70	30	5	32	82
70	31	6	32	32
70	33	6	36	109
70	34	6	36	52
100	50	7	53	158
100	42	6	45	221
100	43	6	45	239
100	39	5	42	198

4.6. Conclusions

We presented a new regular register organization, circular FIFO, as an alternative to separated register organization and to register file organization. Grouping registers into circular FIFOs reduces the control hardware without increasing very much the number of storage elements. In comparison with register file organization, FIFO organization eliminates the overhead of address generation, decoding hardware and the extra access delay. The efficiency of the approach was tested on benchmarks and on complex random examples. Using a complex practical problem like the FIFO allocation problem, we found that the constraint logic programming based on relational interval constraint is a very promising paradigm for solving other CAD problems.

Chapitre 5.

Conclusion

5.1. Résumé des contributions

Nous nous sommes intéressé dans cette thèse à des problemes liés à l'automatisation du processus de synthèse de haut-niveau (SHN) de circuits numériques. Nous avons présenté un ensemble de méthodes algorithmiques qui servent à évaluer la performance de différentes réalisations au niveau transfert de registres d'un circuit. Ces méthodes permettent de calculer des bornes inférieures et supérieures sur la performance que peut atteindre un circuit étant données des contraintes sur les ressources matérielles. Elles se basent sur des techniques d'ordonnancement, telles que la relaxation des contraintes et le "pipeline", et d'analyse du graphe de contrôle et de flot de données (GCFD) modélisant le comportement du circuit. Nous avons d'abord considéré des GCFDs contenant un seul niveau de contrôle (les boucles infinies dans les circuits itératifs). Par la suite, nous avons considéré des GCFDs ayant de structures plus complexes telles que les boucles imbriquées et les branchements conditionnels.

Nous avons montré que tous les algorithmes que nous avons développé ont une faible complexité en temps de calcul. Nous avons aussi réduit la complexité de l'algorithme proposé par Rim *et al* [RIM94] pour déterminer une borne supérieure sur la performance d'un circuit sous des contraintes de ressources. La faible complexité des algorithmes d'estimation est nécessaire en pratique, puisque ces algorithmes vont être exécutés plusieurs fois au cours de l'exploration de l'espace de réalisations.

Une caractéristique de ces méthodes algorithmiques est la facilité de leur intégration dans des systèmes de synthèse industriels. En effet, le modèle de CGFD sur lequel se base

ces méthodes est le modèle standard (ou presque) utilisé pour représenter le comportement des circuits orientés traitement de données. Nous avons déjà intégré un prototype de ces méthodes dans le système de synthèse SIR/CASTLE [CAM96] développé par une autre équipe de recherche. Evidement, une utilisation pratique et efficace de ces méthodes d'estimation nécessite leur adaptation aux types d'applications et de circuits adressés. Cependant, les techniques de base présentées restent les mêmes.

En plus de l'estimation de la performance, les méthodes de calcul de bornes inférieures et supérieures peuvent être utilisées pour développer des algorithmes d'ordonnancement exacts ou heuristiques. Par exemple, l'utilisation des bornes inférieures est immédiate dans un algorithme d'ordonnancement de type "branch and bound".

Dans cette thèse, nous avons présenté aussi une approche efficace pour résoudre le problème d'allocation de registres posé par la SHN des circuits dédiés à des applications spécifiques. Cette approche se base sur une nouvelle organisation de registres sous formes de files circulaires et s'adapte en particulier aux circuits itératifs. Les résultats expérimentaux ont montré qu'une telle organisation permet de réduire le nombre de registres dans le circuit tout en simplifiant la structure du chemins de données. En résolvant efficacement un problème aussi complexe que l'allocation des files circulaires aux variables, nous avons montré que la programmation avec contraintes logiques et à arithmétique des intervalles est un paradigme assez prometteur pour la résolution de certains problèmes d'optimisation dans le domaine de CAO. Les deux principaux avantages de ce paradigme sont: la facilité de décrire certains problèmes d'optimisation en utilisant l'arithmétique des intervalles, et la puissance du procédé de résolution basé sur le rétrécissement répétitif des intervalles.

Cette thèse a aussi contribué à la compréhension des défis et des problèmes posés par la synthèse de haut-niveau et en particulier par la synthèse des circuits réalisant des traitements itératifs.

5.2. Avenues de recherche

Plusieurs avenues de recherche peuvent s'ouvrir comme suite logique de ce travail. Nous les identifions ci-dessous.

Les méthodes d'estimation que nous avons développées permettent de calculer des bornes inférieures et supérieures sur la performance d'un circuit sous des contraintes de ressources. Une suite de ce travail serait de développer des méthodes pour résoudre le problème dual: déterminer des bornes inférieures et supérieures sur les ressources sous des contraintes de performance. Dans le cas particulier où le comportement du circuit ne contient pas des branchements conditionnels, nos méthodes s'adaptent facilement pour résoudre le problème dual. Par contre dans le cas général, d'autres techniques doivent être développées.

Comme nous l'avons déjà dit dans la section précédente, les méthodes de calcul des bornes inférieures et supérieures sur la performance peuvent être utilisées pour développer des algorithmes d'ordonnancement exacts ou heuristiques. En particulier des algorithmes d'ordonnancement de type pipeline, qui posent un grand défi dans la synthèse des circuits itératifs dédiés à des applications de traitements d'images et de signaux.

Bien que nous avons apporté des solutions satisfaisantes au problème de calcul des bornes inférieures et supérieures sur la performance, ce problème se prête encore à des améliorations: la précision des bornes peuvent être améliorée et/ou la complexité des algorithmes peut être réduite.

L'approche d'organiser les registres sous forme de files circulaires parait très prometteuse. Rappelons que les deux critères que nous avons optimisé durant l'allocation des files circulaires aux variables sont le nombre de files circulaires et le nombre total de registres dans ces files. Or les mouvements de données (décalage et rotation) dans les files augmentent la consommation d'énergie. La première suite de ce travail serait de reformuler le problème d'allocation des file circulaires en incluant le critère d'optimisation du nombre de mouvements de données dans le files, puis de résoudre ce problème. La deuxième suite de ce travail serait de développer des algorithmes

d'allocation qui combine les deux types d'organisation: les files circulaires et les banques de registres.

Bibliographie

[AHO83] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Data Structures and Algorithms*, Wesley Publishing Company, MA, 1983.

[AIK88] A. Aiken, A. Nicolau, "Optimal loop Parallelization" in proceedings of the *SIGPLAN*, Vol. 23, N. 7, 1988.

[AIK91] A. Aiken, A. Nicolau, "A realistic Resource-Constrained Software Pipelining Algorithm", *Advances in Languages and Compilers for Parallel Processing*, MIT Press, 1991.

[ALM90] M. A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Cost", *IEEE tran. on Software Engineering*, Vol. 16, December 1990.

[ALO94] M. Aloqeely, C. Y. Roger Chen, "Sequencer-Based Data Path Synthesis of Regular Iterative Algorithms", in proceedings of the 31*st Design Automation Conference*, 1994.

[BEN96a] I.E. Bennour, E.M. Aboulhamid, "Lower-bounds on the Iteration and Initiation Interval of functional Pipelining and Loop Folding," the journal *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 1, No. 4, 1996.

[BEN96b] I.E. Bennour, M. Langevin, E.M. Aboulhamid, "Performance Analysis for Hardware/Software Cosynthesis", in proceedings of the *Canadian Conference on Electrical and Computer Engineering, 1996.*

[BEN96c] I.E. Bennour, E.M. Aboulhamid, "Register Allocation using Circular FIFOs", in proceedings of the *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1996.

[BEN96d] I.E. Bennour, E.M. Aboulhamid, "Les problèmes d'ordonnancement cyclique dans la synthèse de circuits numériques", submitted to the journal *Technique et Science Informatique.*

[BNR93] *BNR PROLOG, User Guide,* Version 4, Ottawa, Canada, 1993.

[CAM96] R. Camposano, J. Wilberg, "Embedded System Design", *Design Automation for Embedded Systems*, Vol. 1, pp. 5-50, 1996.

[CAR88] P. Carlier, P. Chrétienne, *Les Problèmes d'Ordonnancement: Modélisation/ Complexité Algorithmes* , Masson, Paris 1988.

[CHA93a] L.-F. Chao, E. H.-M. Sha, "Rate-Optimal Static Scheduling for DSP Data-Flow Programs", in proceedings of the *Third Great Lakes Symposium on VLSI*, March 1993.

[CHA93b] L.-F. Chao, A. LaPaugh, "Rotation Scheduling: a Loop Pipelining Algorithm", in proceedings of the *Design Automation Conference* 1993.

[CHA94] S. Chaudhuri, R. A. Walker, "Computing Lower Bounds on Functional Units Before Scheduling" in proceedings of the *7th Intern. Symp. on High Level Synthesis*, 1994.

[CHE91] C.Y. Chen, M. Morcz, "A Delay Distribution Methodology for the Optimal Systolic Synthesis of Linear Recurrence Algorithms", *IEEE Trans. on Computer-Aided Design*, Vol. 10, pp. 685-697, June 1991.

[CHR91] P. Chrétienne, "The Basic Cyclic Scheduling Problem With Deadlines", *Discrete Applied Mathematics*, Vol. 30, 1991.

[CHR83] P. Chrétienne, *Les rèsaux de Petri Temporisés*, Thèse d'état, Université Pierre et Marie Curie, 1983.

[COR90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press and Mc Graw Hill, 1990.

[DEM94] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.

[DON92] H. V. Dongen, G. R. Gao, Q. Ning, "A Polynomial Time Method for Optimal Software Pipelining", Parallel processing, CONPAR VAPPV 92, *Lectures Notes in Computer Sciences*, Vol. 634, 1992.

[GAJ92] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis*, Kluwer Academic Publishers, Boston, 1992.

[GAS92] F. Gasperoni, U. Schwiegelshohn, "Scheduling Loops on Parallel Processors:

A Simple Algorithm with Close to Optimum Performance", Parallel Processing, CONPAR VAPPV 92, *Lectures Notes in Computer Sciences*, Vol. 634, 1992.

[GER92] H. Gerez, S. M. H de Groot, O. E. Herrmann, "A Polynomial-time Algorithm for the Computation of the Iterative-period Bound in Recursive Data-Flow Graphs", *IEEE Trans. on Circuits and Systems*, Vol. 39, No. 1, January. 1992.

[GON94] J. Gong, D. D. Gajski, A. Nicolau, "A Performance Evaluator for Parameterized ASIC Architectures", in proceedings of the *European Design Automation Conference*, 1994.

[GOO90] G. Goossens, J. Rabaey, J. Vandewalle, H. De Man, "An efficient Microcode Compiler for Application Specific DSP Processors", *IEEE Trans. on Computer-Aided Design*, Vol. 9, No. 9, September 1990.

[GRO92] M. H de Groot, S. H. Gerez, O. E. Herrmann, "Range-chart-guided Iterative Data Flow Graph Scheduling", *IEEE Trans. on Circuits and Systems*, Vol. 39, No. 5, May 1992.

[HAN90] C. Hanen, "Les Tables de Réservation Numériques: Un Outil pour la Résolution de Certains Problèmes D'ordonnancement Cycliques", *Recherche Opérationnelle,* Vol. 24, No. 2, 1990.

[HAN94] C. Hanen, "Study of NP-hard Cyclic Scheduling Problem: the Recurrent Job-Shop", *European Journal of Operation Research*, Vol. 72, 1994.

[HAN95] C. Hanen, A. Munier "A Study of the Cyclic Scheduling Problem on Parallel Processors", *Discrete Applied Mathematics* , Vol. 57, 1995.

[HEN92] L.Hendre, G.R. Gao, E. Altman, C. Mukerji, "A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs", *Lectures Notes in Computer Sciences* , Vol. 641, 1992.

[HU61] T. C. Hu, "Parallel Sequencing and Assembly Line Problems" *Operations Research*, Vol. 9, pp. 841-848, 1961.

[HU93] Y. Hu, A. Ghouse, B. S. Carlson, "Lower Bounds on the Iteration Time and the Number of Resources for Functional Pipelined Data Flow Graphs" in proceedings of the *International Conference on Computer Design*, 1993.

[HWA93] C. T. Hwang, Y. C. Hsu, Y. L. Lin, "PLS: A Scheduler for Pipeline Synthesis",

IEEE Trans. on Computer-Aided Design, Vol. 12, September 1993.

[HWA91] C. T. Hwang, J. H. Lee, Y. C. Hsu, "A Formal Approach to the Scheduling Problem in High Level Synthesis" *IEEE Trans. on Computer-Aided Design*, Vol. 10, pp. 669-683, April 1991.

[IWA90] K. Iwano, S. Yeh, "An Efficient Algorithm for Optimal Loop Parallelization", Algorithms, *Lectures Notes in Computer Sciences*, Vol. 450, 1990.

[JAI92] R. Jain, A. C. Parker, N. Parker, "Predicting System-level Area and Delay for Pipelined and non-pipelined Designs" *IEEE Trans. on Computer-Aided Design*, Vol. 11, August 1992.

[JEN94]L.-G. Jeng, L.-G. Chen, "Rate-optimal DSP Synthesis by Pipeline and Minimum Unfolding", *IEEE Trans. on VLSI*, Vol. 2, No. 1, March 1994.

[KAR78] R.M. Karp, " A Characterization of the Minimum Cycle in a Digraph" Discrete *Mathematics* Vol. 23, 1978.

[KOG81] P. M. Kogge, "*The architecture of Pipe-lined Computers*", McGraw Hill, New York, 1981.

[KUN95] S. Y. Kung, H.J. Witehouse, T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, pp. 258-264, 1985.

[LAM88] M. Lam, "Software Pipelining: an Effective Scheduling Technique for VLIW Machines" in proceedings of the *SIGPLAN* 88 *Conference on Prog. Language Design and Implementation*, 1988.

[LAN93] M. Langevin, E. Cerny, "A Recursive Technique for Computing Lower-Bound Performance of Schedules" in proceedings of the *International Conference on Computer Design*, 1993.

[LEE94] T.-F. Lee, A. C.-H. Wu, D. D. Gajski, "A Transformation-based Method for Loop Folding", *IEEE Trans. on Computer-Aided Design*, Vol. 13, No. 4, April 1994.

[LEI91] C. E. Leiserson, J. B. Saxe, "Retiming Synchronous Circuitry", *Algorithmica*, Vol. 6, 1991.

[LI95] Y.-T. S. Li, S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", 32nd *Design Automation Conference*, 1995.

[LIU89]D. Liu, W. Giloi, "A Loop Optimization Technique Based on Scheduling Table",

in proceedings of the 22nd *Annual int. Workshop on Microprogramming and Microarchitecture*, 1989.

[LUC93] L. E. Lucke, K. K. Parhi, "Data-flow Transformations for Critical Path Time Reduction in High Level DSP Synthesis" *IEEE Trans. on Computer-Aided Design*, Vol. 12, pp. 1063-1068, July. 1993.

[MIC92] P. Michel, U. Lauther, P. Dusy, *The Synthesis Approach to Digital System Design*, Kluwer Academic Publishers, Boston, 1992.

[MOO79] R.E. Moore, *Methods and Applications of Interval Analysis*, SIAM, 1979.

[MUN91] A. Munier, "Résolution d'un Problème d'Ordonnancement Cylique à Itérations Indépendantes et Contraintes de Ressources", *Recherche Opérationnelle*, Vol. 25, No. 2, 1991.

[PAR91] K. K. Parhi, D. G. Messerschmitt, "Static Rate-optimal Scheduling of Iterative Data-flow Programs Via Optimum Unfolding", *IEEE Trans. on Computers*, Vol. 40, No.2, Feb. 1991.

[PAR88] N. Park, A. C. Parker, "Sehwa: a Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on Computer-Aided Design*, Vol. 7, No. 3, March 1988.

[PAR93] I.-C. Park, C.-M. Kyung, "FAMOS: An Efficient Scheduling Algorithm for High-Level Synthesis" *IEEE Trans. on Computer-Aided Design*, Vol. 12, pp. 1437-1448, October. 1993.

[PAU89] P. G. Paulin, J. P. Knight, "Force-directed Scheduling for the Behavioral Synthesis of ASIC's" *IEEE Trans. on Computer-Aided Design*, Vol. 8, June 1989.

[PAS94] N. L. Passos, E. H. Sha, S. C. Bass, "Loop Pipelining for Scheduling Multi-Dimensional System via Rotation", in proceedings of the *31th Design Automation Conference*, 1994.

[POT90] R.R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation Based Synthesis" in proceedings of the *27th Design Automation Conference*, 1990.

[POT94] M. Potkonjak, J. Rabaey, "Optimizing Throughput and Resource Utilization using Pipelining: Transformation Based Approach" *Journal of VLSI Signal Processing*, Vol. 8, pp. 117-130, 1994.

[POT94a] M. Potkonjak, J. Rabaey, "Optimizing Resource Utilization using Transformations", *IEEE Trans. on Computer-Aided Design*, Vol. 13, pp. 277-292, Marsh 1994.

[RAB94] J. Rabaey, M. Potkonjak, "Estimating Implementation Bounds for Real Time DSP Application Specific Circuits",*IEEE Trans. on Computer-Aided Design*, Vol. 13, June 1994.

[RAU92] B. R. Rau, M. Lee, P. P. Tirumalai, M. S. Schlansker, "Register Allocation for Modulo Scheduled Loops: Strategies, Algorithms and Heuristics", in proceedings of the *SIGPLAN* 92 *Conference on Programming Language Design and Implementation*, 1992.

[RIM94] M. Rim, R. Jain, "Lower-bound Performance Estimation for the High-Level Synthesis Scheduling Problem", *IEEE Trans. on Computer-Aided Design*, Vol. 13, pp 81-88, 1994.

[ROU92] R. Roundy, "Cyclic Schedules For Job Shops with Identical Jobs", *Mathematics of operations research*, Vol. 17, No. 4, pp 842-865, November 1992.

[SCH85] D. A. Schwartz, T. P. Barnwell , "Cyclo-Static Multiprocessor Scheduling For The Optimal Realization of Shift-Invariant Flow Graphs", in proceedings of the *International Conference on Accoustics, Speech and Signal Processing*, 1985.

[SCH89] U. Schwiegelshohn, F. Gasperoni, K. Ebcioglu, "On Optimal Loop Parallelisation", 22nd *Annual int. Workshop on Microprogramming and Microarchitecture*, 1989.

[SHA93] A. Sharma, R. Jain, "Estimation Architectural Resources and Performance for High-Level Synthesis Applications", *IEEE trans. on VLSI*, Vol. 1, pp. 175-190, 1993.

[SRI94] M. B. Srivatava, M. Potkonjak "Transforming Linear Systems for Joint Latency and Throughput Optimisztion", in proceedings of the *European Design Automation Conference*, pp. 267-271, 1994.

[SU97]B. Su, S. Ding, J. Wang, J. Xia, "GURPR--A Method for Global Software Pipelining", in proceedings of the 20*th Annual Workshop on Microprogramming*, 1987.

[WAN93] C.-Y. Wang, K. K. Parhi, "Loop List Scheduler for DSP Algorithms Under Resource Constraints", in proceedings of the *IEEE International Symposium on*

Circuits and Systems (ISCAS) 1993.

[WOL91] M. E. Wolf, M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE trans. on Parallel and Distributed Systems*, Vol. 2, No. 4, October 1991.

[ZAK89] A. Zaky, P. Sadayappan, "Optimal Static Scheduling of Sequential Loops on Multiprocessors", in proceedings of the *International Conference on Parallel Processing*, Vol. III, 1989.