D.E.A. INFORMATIQUE

1993-1994

Étude du parallélisme d'instructions

Étude bibliographique
et
Rapport de stage

Erven Rohou

Stage effectué à :
Irisa / Équipe Caps

Sous la responsabilité de :
Nathalie Drach et André Seznec

Résumé

La recherche en informatique attache aujourd'hui un grand intérêt au parallélisme pour accroître les performances des ordinateurs. La synthèse bibliographique s'intéresse plus particulièrement au parallélisme d'instructions. Après avoir présenté quelques architectures classiques de processeurs, elle dresse un état de l'art des principales techniques logicielles et matérielles utilisées pour étudier, exploiter et accroître significativement le degré de parallélisme d'instructions d'un programme.

La deuxième partie présente ensuite le travail réalisé au cours du stage. Quelques précisions sur les aléas sont apportées puis le développement d'un simulateur de processeur superscalaire est exposé. Ce simulateur a permis de comparer plusieurs structures de pipelines et d'étudier l'impact de différents paramètres comme la prédiction de branchement, le temps d'accès à la mémoire, etc. Les résultats et les conclusions qui en découlent sont présentés dans le dernier chapitre.

Mots clés : parallélisme d'instructions, régularité du parallélisme, réordonnancement, prédiction de branchement, optimisation de boucles, renommage, séquencement, aléas.

Table des matières

Table des figures

Liste des tableaux

Première partie

Étude bibliographique

Introduction

Plusieurs solutions existent pour accélérer la vitesse d'exécution d'un programme. La première consiste à le diviser en plusieurs parties s'exécutant chacune sur un processeur différent. Ce problème ne sera pas abordé ici : on ne s'intéresse qu'à un seul processeur exécutant un seul flot d'instructions. Il faut donc soit diminuer le temps de cycle du processeur, soit augmenter la quantité de traitement effectué par le processeur à chaque cycle, c'est-à-dire mettre en œuvre du parallélisme. On peut distinguer trois sortes de parallélisme potentiel :

- parallélisme vectoriel : c'est le parallélisme le plus évident, celui que l'on rencontre lorsque l'on calcule la somme de deux vecteurs. Il est clair que chaque composante du vecteur-somme peut être obtenue indépendamment des autres. Tous les calculs peuvent être effectués simultanément.
- parallélisme de boucle : si dans une boucle les traitements effectués à chaque itération sont indépendants, alors on peut les mener de front. La difficulté consiste à détecter, quand le corps de la boucle est relativement complexe, s'il y a bien indépendance.
- parallélisme d'instructions : de manière générale, si deux portions de code d'une application ne dépendent pas l'une de l'autre, il est possible de les exécuter en parallèle. Ceci est valable aussi bien pour des instructions successives, que pour des blocs séparés, voire pour des processus distincts, qu'ils appartiennent ou non à la même application. Au niveau des micro-instructions : il est intéressant de pouvoir utiliser au maximum les unités fonctionnelles du processeur et si rien ne s'y oppose, on souhaite en utiliser le plus possible en même temps. Pour cela on développe par exemple des architectures pipelines, superpipelines ou superscalaires.

C'est de cette dernière sorte de parallélisme qu'il sera question ici.

Des études théoriques ont été menées et le développement rapide de la technologie a permis d'aboutir à des résultats concrets. En effet sur le marché existent actuellement des machines qui mettent en œuvre un certain degré de parallélisme : machines pipelines (voire superpipelines), VLIW (pour *very long instruction word*) ou superscalaires.

La question qui se pose maintenant est la suivante : est-il réellement intéressant de construire des machines dotées de telles performances ? Plus précisément : y a-t-il vraiment tant de parallélisme potentiel dans nos programmes pour que l'on ait besoin d'utiliser des machines si complexes ? Si de façon intrinsèque et pour des raisons de dépendance logique entre instructions, le degré de parallélisme ne peut dépasser quelques unités, alors il est parfaitement inutile d'utiliser des dizaines d'unités fonctionnelles. La plus grande partie d'entre elles sera toujours en attente du résultat d'une autre pour pouvoir travailler et donc sous-utilisée.

Le chapitre 1 présente les architecture existantes pour exploiter le parallélisme d'instructions. Le chapitre 2 expose les méthodes utilisées pour accroître et/ou exhiber le parallélisme. Les modèles d'ordinateurs, les paramètres des simulations et les résultats sont développés dans le chapitre 3.

Chapitre 1

Architectures de processeurs

L'architecture des machines existantes influe fortement sur le type de parallélisme que l'on souhaite exhiber et sur les méthodes qu'il est raisonable de développer pour y parvenir.

1.1 Architecture pipeline

L'idée est de considérer une instruction comme une succession d'actions élémentaires du processeur : *chargement, décodage, exécution,* etc. (cf. figure 1.1). Le temps de cycle du processeur est le temps nécessaire à l'exécution de la plus longue de ces phases. À chaque cycle l'instruction progresse d'un seul étage dans le pipeline. Il devient possible de lancer une instruction à chaque cycle, puisque les unités fonctionnelles requises sont différentes pour chaque phase. Le degré de

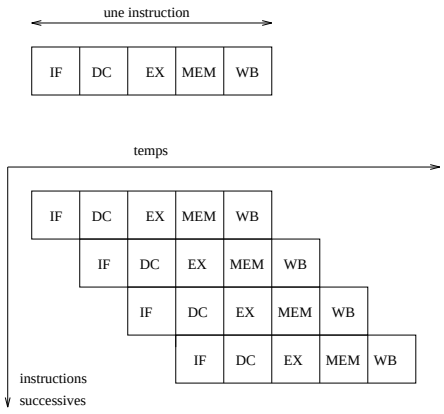

FIG. 1.1 – Pipeline d'instructions

parallélisme potentiel que le processeur peut exploiter est égal au nombre d'étages du pipeline.

Il est possible d'augmenter le nombre d'étages pour tenter d'accroître les performances de la machine[1], mais seulement jusqu'à une certaine limite. En effet il faut ajouter des tampons (*buffers*) entre les étages successifs du pipeline, ce qui est coûteux en temps (quelques nanosecondes). Si l'on découpe une instruction s'exécutant en un temps t sur une machine non-pipeline en d sous-instructions et que l'on rajoute $d-1$ tampons de temps d'accès τ, le temps d'exécution d'une instruction avec pipeline sera de $d \times t/d + (d-1)\tau = t + (d-1)\tau$. Comme il est possible de lancer d fois plus d'instructions, le gain maximal en performance est :

$$S_d = \frac{t}{\frac{t+(d-1)\tau}{d}} = \frac{td}{t+(d-1)\tau}$$

Cette quantité est croissante et a pour limite quand d devient grand :

$$\lim_{d \to +\infty} S_d = \frac{t}{\tau}$$

[1] Certains parlent alors de superpipeline.

1.2 Architectures superscalaires et VLIW

L'approche est différente pour les machines superscalaires et VLIW. Ces deux types de machines sont capables de lancer plus d'une instruction à la fois. Ces instructions sont exécutées réellement en parallèle.

Machines superscalaires
Une machine superscalaire de degré n peut exécuter simultanément n instructions. Pour être utilisée au maximum de ses capacités, elle requiert à tout instant un degré de parallélisme d'instructions au moins égal à n.

Dans un superscalaire *symétrique* toutes les unités fonctionnelles sont dupliquées et toutes les combinaisons d'instructions sont *a priori* autorisées. À l'opposé dans un superscalaire *naturel* aucune unité n'est dupliquée. Les instructions sont attribuées aux unités fonctionnelles quand celles-ci sont libres et le nombre de combinaisons possibles est fortement réduit. Le DEC 21064 est un exemple de superscalaire naturel. Toutes les solutions intermédiaires sont évidemment possibles : le TI SuperSparc par exemple peut exécuter deux opérations ALU (unité arithmétique et logique) indépendantes par cycle. Les architectures du DEC 21064 et du TI SuperSparc sont détaillées dans [SKV93].

Machines VLIW
Comme leur nom l'indique, les machines VLIW (*very long instruction word*) ont des mots d'instructions relativement longs : de l'ordre de quelques centaines de bits. Le mot est divisé en champs de taille fixe qui commandent chacun une unité fonctionnelle. À chaque cycle du processeur, toutes les unités sont commandées. Elles reçoivent éventuellement un ordre vide s'il n'y a rien à faire.

Comparaison
Les architectures superscalaires et VLIW sont très semblables. Il existe toutefois trois différences ([JW89]) :

- Le décodage des instructions est facilité dans une machine VLIW dans la mesure où le mot a un format fixe, connu d'avance. Le matériel n'a aucune vérification à faire sur les éventuelles dépendances de données. Tout est déterminé par le compilateur. À l'inverse, ces dépendances doivent être détectées par un processeur superscalaire au moment de l'exécution, ce qui implique un matériel beaucoup plus complexe.
- Le format du mot d'instruction d'une machine VLIW étant constant, beaucoup de code inutile est généré quand le degré de parallélisme est insuffisant et nombre d'unités fonctionnelles restent inactives.
- Le code objet des machines superscalaires peut être compatible avec des machines non-parallèles, ce qui n'est pas le cas des machines VLIW. Le degré de parallélisme exploitable par une machine VLIW dépend fortement de son architecture et le code objet n'est absolument pas portable.

Les résultats des simulations faites par Jouppi et Wall dans [JW89] montrent que les performances des machines pipelines sont légèrement inférieures à celles des superscalaires, mais que la différence tend vers zéro lorsque le degré (nombre d'étages du pipeline ou d'instructions exécutées en parallèle) augmente. Ce léger avantage est toutefois compensé par le surcoût de mise en œuvre du superscalaire.

Chapitre 2

Différentes formes de parallélisme d'instructions

Les simulations montrent que le parallélisme d'instructions existe de façon implicite dans tous les programmes testés ([Wal91, LW92, BYP+91]), mais que certains types d'applications ont une aptitude à présenter un degré de parallélisme plus élevé. Les applications de calcul scientifique en particulier ont tendance à manipuler de grands tableaux ou à multiplier des matrices, tâches qui se prêtent bien à une parallélisation. On peut trouver une étude du parallélisme d'instructions des applications non-scientifiques dans [SJH89].

La loi d'Amdahl exprime l'accélération (*speedup*) qu'il est possible d'obtenir en parallélisant certaines portions d'un programme. Si p est le nombre de processeurs utilisables ou d'unités fonctionnant en parallèle et α la proportion du code susceptible d'être exécutée en parallèle, alors l'accélération potentielle est donnée par :

$$S_p = \frac{1}{1 + \alpha(\frac{1}{p} - 1)}$$

Or on a $\lim_{p \to \infty} S_p = \frac{1}{1 - \alpha}$. L'accélération maximale est majorée par une constante qui dépend du parallélisme intrinsèque du code à exécuter.

2.1 Définition du parallélisme d'instructions - Problèmes posés

2.1.1 Définition

La mise en œuvre du parallélisme correspond à l'exécution simultanée de plusieurs instructions. Il est nécessaire que ces instructions soient logiquement indépendantes, faute de quoi l'exécution séquentielle sera obligatoire. La difficulté majeure est de détecter que des instructions ou des blocs d'instructions sont indépendants ou que les dépendances sont artificielles et qu'elles ont été générées à la compilation. La figure 2.1 montre un exemple de parallélisme d'instructions potentiel. Le fragment de code est composé de trois instructions qui peuvent parfaitement être exécutées en même temps si `r4+15`$\neq$`r5`. Le parallélisme d'instructions est égal au rapport du nombre d'instructions exécutées par le nombre de cycles requis. Dans le cas de la figure 2.1, il est égal à trois. Des études suggèrent que le parallélisme potentiel d'un bloc excède rarement la

```
ld r1, 0[r4]
ld r2, 0[r5]
st 15[r4], r6
```

FIG. 2.1 – Instructions indépendantes

valeur de 4 ([Wal91, SJH89]). Pour Butler et al. [BYP+91], par contre, il est possible d'atteindre

des degrés beaucoup plus élevés si l'on réussit à se débarrasser de certaines contraintes liées au matériel et aux techniques utilisées.

On peut trouver dans [Kri90] un survol de différentes papiers ayant trait à l'optimisation de code et au réordonnancement en vue d'exploiter le parallélisme d'instructions.

Il faut noter que les notions de parallélisme d'instructions et de vitesse d'exécution ne vont pas forcément de pair. Une optimisation peut diminuer le temps d'exécution d'un programme en même temps que le degré de parallélisme. Il est montré dans [JW89] que la majorité des optimisations classiques ont peu d'effet sur le parallélisme d'instructions et ont souvent tendance à le faire diminuer. Ceci s'explique par le fait qu'une optimisation consiste souvent à supprimer des portions de code redondantes ou inutiles qui fournissaient artificiellement du parallélisme potentiel.

2.1.2 Aléas

En plus de la difficulté de trouver assez de parallélisme d'instructions à exploiter pour garder les unités fonctionnelles à un niveau d'utilisation suffisant, de nouvelles contraintes apparaissent dès lors que l'on veut exécuter en parallèle plusieurs instructions. On peut les classer en trois catégories : aléas de structure, de données et de contrôle.

Aléas de structure : il s'agit d'un conflit de ressources. Plusieurs instructions qui s'exécutent en parallèle ont besoin d'accéder à la même unité fonctionnelle (ALU, unité flottante, mémoire, ...). Cet aléa est fonction du matériel utilisé. Il peut apparaître avec un certain processeur et pas avec un autre qui dispose d'un plus grand nombre d'unités.

Aléas de données : il en existe trois : lecture après écriture (*RAW : read after write*), écriture après lecture (*WAR : write after read*) et écriture après écriture (*WAW : write after write*). Seuls les aléas RAW sont de véritables dépendances, comme on le verra dans le paragraphe 2.5.1. Les deux autres peuvent être résolus par renommage. Toutefois des aléas WAW peuvent être à l'origine de difficultés rencontrées dans les pipelines : du fait que des instructions, comme les opérations flottantes, ont des temps de latence plus longs, certaines écritures sont retardées.

Aléas de contrôle : lorsque l'une des instructions est un branchement, il est difficile de savoir si l'instruction immédiatement suivante sera exécutée ou non. Et dans le cas où elle ne l'est pas, le problème est de déterminer l'adresse de la prochaine (par exemple si l'adresse est contenue dans un registre dont la valeur sera calculée au dernier moment).

Il est indispensable que le matériel puisse détecter ces aléas lorsqu'ils surviennent [1], mais il est préférable de les éviter par logiciel puisqu'ils sont la source d'une dégradation des performances. Chaque aléa provoque la mise en attente d'une instruction jusqu'à la résolution du conflit et donc une perte de cycles.

2.2 Régularité

Une fois le degré de parallélisme d'un programme déterminé, il est intéressant de calculer sa *régularité*. Cette quantité est introduite dans [TGH92]. C'est une indication du taux d'utilisation des processeurs ou unités fonctionnelles alloués pour l'exécution du programme. Intuitivement on comprend que si un programme est constitué d'une partie purement séquentielle et d'une partie où le parallélisme est très important, il faudra un grand nombre de processeurs ou d'unités fonctionnelles pour exploiter le maximum de parallélisme dans la seconde partie, mais leur taux d'utilisation sera très faible dans la première. La loi d'Amdahl montre que le gain en performance avec un tel programme sera faible.

Un degré de parallélisme plus faible, mais constant, est plus intéressant.

Le parallélisme exploitable est borné par le nombre d'unités fonctionnelles disponibles quand ce nombre est petit et par le degré de parallélisme maximal du programme quand ce nombre

[1] Le problème ne se pose pas pour les VLIW.

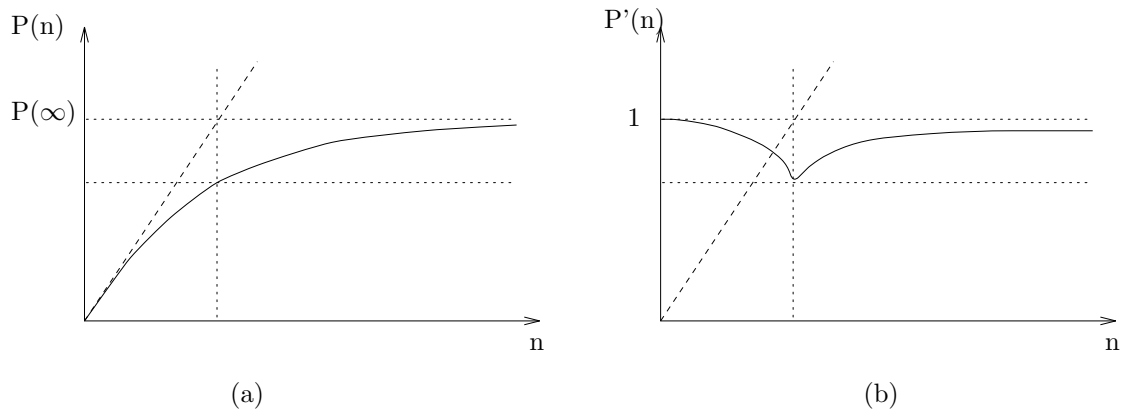

FIG. 2.2 – Parallélisme et régularité

tend vers l'infini, noté $P(\infty)$ (cf. figure 2.2-a). On peut donc le normaliser (cf. figure 2.2-b) :

$$P'(n) = \begin{cases} \frac{P(n)}{n} & si\ n \leq P(\infty) \\ \\ \frac{P(n)}{P(\infty)} & si\ n > P(\infty) \end{cases}$$

Le nombre d'unités nécessaires pour exploiter pleinement le parallélisme est un nombre entier, c'est-à-dire au minimum $\lceil P(\infty) \rceil$. $P'(n)$ passe par un minimum pour une valeur proche de $n = P(\infty)$. La *régularité* est définie comme la valeur du parallélisme normalisée en ce point exprimée en pourcentage :

$$S = P'(\lceil P(\infty) \rceil) = \frac{P(\lceil P(\infty) \rceil)}{P(\infty)}$$

Pour calculer cette valeur on procède en deux étapes :

1. On détermine $P(\infty)$ en simulant l'exécution du programme sur une machine disposant d'une infinité de ressources.
2. On simule le programme une deuxième fois en supposant le nombre de ressources disponibles égal à la valeur trouvée précédemment.

Une valeur de S proche de 1 correspond à une bonne utilisation des ressources tandis qu'une valeur proche de 0 est un signe que le parallélisme est mal réparti et que les ressources disponibles sont sous-utilisées.

2.3 Augmentation du degré de parallélisme d'instructions

Il est possible d'augmenter le degré de parallélisme d'un segment de code en réorganisant les instructions au sein d'un corps de boucle. Deux techniques spécifiques réalisent ces migrations de code : le *dépliage des boucles* et le *pipeline logiciel*. Une technique plus générale connue sous le nom de *compactage de micro-code* est développée dans [Fis81]. Elle opère sur les micro-codes et permet de les rendre plus compacts, donc plus efficaces.

2.3.1 Dépliage des boucles

Le *dépliage des boucles* ([Wal91, HP90]) est une technique d'optimisation utilisée par les compilateurs. Elle consiste à réécrire une boucle en recopiant son corps un certain nombre de fois de façon à ce que le nombre d'itérations soit plus faible mais que chaque itération du code modifié fasse le travail de plusieurs itérations du code original. La figure 2.3 montre un exemple de réécriture. Le code « déplié » contient moins d'itérations que le code original. En conséquence les temps d'initialisation et de terminaison de la boucle sont réduits. En outre il est plus facile de détecter du parallélisme potentiel dans le code transformé, du fait de sa linéarité.

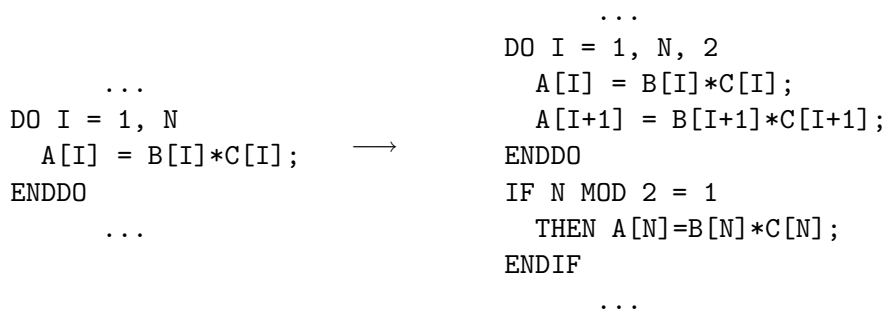

FIG. 2.3 – Dépliage de boucle 2 fois

On peut ensuite transformer le nouveau corps de boucle pour s'affranchir des temps de latence de certaines instructions. Cette méthode a l'inconvénient de provoquer une expansion du code s'il y a beaucoup de boucles ou si elles sont « dépliées » un grand nombre de fois.

2.3.2 Pipeline logiciel

Le *pipeline logiciel* est une technique de compilation qui consiste à déplacer des instructions dans le corps d'une boucle. Une instruction est déplacée d'une itération dans la suivante ou la précédente de manière à augmenter le parallélisme ([HP90]). Le but est qu'une itération du code transformé soit constituée de séquences d'instructions provenant d'itérations différentes du code original. On retrouve ainsi une partie du parallélisme que l'on aurait en dépliant la boucle, mais on évite l'inconvénient de l'expansion du code.

On peut trouver un algorithme de pipeline logiciel dans [Bod89], ainsi qu'une généralisation de la méthode aux boucles récurrentes. L'algorithme est accompagné de justifications théoriques et d'une analyse de performances. Un algorithme de dépliage de boucles pour les pipelines logiciels est également présenté : le pipeline logiciel construit est équivalent à un dépliage complet de la boucle (sauf pour les premières itérations), il tient compte des ressources et peut traiter les boucles récurrentes.

Pipeline logiciel décomposé
Une variante est exposée dans [WE93] : la technique du pipeline logiciel est vue comme une transformation d'un vecteur colonne d'instructions à une dimension en une matrice à deux dimensions. Les lignes représentent les cycles où sont exécutées les opérations et les colonnes indiquent l'itération dans la boucle d'origine. Le travail à effectuer se décompose donc naturellement en deux parties : calcul du numéro de ligne et calcul du numéro de colonne.

L'article présente d'abord une approche théorique, puis deux algorithmes de complexité $O(n^3)$.

2.3.3 Compactage de micro-code

Le *compactage de micro-code* est une façon de rendre parallèle (horizontal) un code séquentiel (vertical). Il permet d'obtenir un code plus efficace. Le principe général est de déplacer des instructions et de les regrouper en blocs de telle façon que le taux d'utilisation des unités fonctionnelles soit maximal pour chaque bloc à tout instant et que le nombre de conflits pour l'utilisation de ces unités soit minimal. Le déplacement d'instructions doit bien évidemment respecter la sémantique du programme.

Deux approches sont possibles : compactage local ou global.

Compactage local : le code est divisé en blocs de telle façon que toutes les instructions d'un bloc soient compatibles, c'est-à-dire qu'aucun conflit de ressources n'existe au sein des blocs. Ces blocs seront exécutés selon un ordre qui respecte les dépendances, mais l'algorithme de compactage ne s'intéresse qu'aux blocs indépendamment les uns des autres. Un

graphe de dépendance est construit pour le bloc étudié en considérant l'ordre partiel $\prec$ sur les instructions : m_i et m_j étant deux instructions, on a $m_i \prec m_j$ si l'une des deux conditions suivantes est réalisée :
- m_i écrit dans un registre qui est lu par m_j. Il s'agit d'une dépendance de type RAW.
- m_i lit un registre et m_j est la première des instructions suivantes qui écrit dans ce registre. C'est une dépendance de type WAR.

Le graphe acyclique orienté (*DAG : directed acyclic graph*) contient la sémantique à respecter lors du choix de l'ordre de séquencement des instructions. Le problème du choix de cet ordre est abordé dans le paragraphe 2.3.4.

Compactage global : le principal défaut du compactage local est qu'il est incapable d'extraire tout le parallélisme présent dans un fragment de code. En effet des expériences ont montré que la majorité du parallélisme se trouve entre les blocs : les résultats de Theobald, Gao et Hendren dans [TGH92] par exemple indiquent qu'une fenêtre d'instructions trop petite réduit considérablement le degré de parallélisme exploitable. Et comme les blocs ont tendance à être très courts dans les micro-codes, la méthode n'est pas entièrement satisfaisante. Le compactage global préfère considérer le code dans son ensemble pour effectuer les migrations d'instructions nécessaires.

Une technique appelée *ordonnancement de trace* est présentée dans [Fis81]. Une trace est une séquence d'instructions successives (sans boucle) que l'on peut trouver dans une exécution particulière du programme et pour un certain jeu de données. L'algorithme de compactage travaille avec les traces sans se préoccuper des blocs : il construit tout d'abord le DAG correspondant puis essaie de réorganiser les micro-instructions. Cette méthode permet d'avoir une vue plus générale du code à optimiser et de réaliser des migrations de code plus pertinentes que n'aurait pu le faire un compactage local.

2.3.4 Existence et complexité

La réorganisation de code à la compilation est un problème qui a été étudié de façon très théorique dans [HG83]. Le cas général a été prouvé $\mathcal{NP}$-complet. Cependant la pratique montre que des heuristiques simples donnent des résultats suffisamment bons pour que l'on puisse négliger leur non-optimalité.

Fisher ([Fis81]) propose une heuristique : *ordonnancement par liste de micro-instructions*. Des priorités sont attribuées aux instructions. Parmi toutes celles dont les prédécesseurs ont déjà été séquencées et qui ne vont pas entrer en conflit de ressources avec celles déjà choisies pour le cycle courant, on sélectionne celle qui a la plus forte priorité. Une solution possible est d'attribuer les priorités en fonction de la longueur de la plus longue chaîne du DAG commençant à ce point.

Une autre heuristique est exposée dans [HG83]. Son efficacité et ses propriétés y sont étudiées.

L'existence d'une solution au problème d'ordonnancement de micro-code est prouvée sous certaines hypothèses dans [Bod89]. Un algorithme de compactage de complexité $O(n^2)$ utilisant aussi l'ordonnancement par liste est étudié.

2.4 Cas des branchements

2.4.1 Prédiction de branchement et de saut

Lorsque l'on souhaite exécuter simultanément du code appartenant à plusieurs blocs distincts de code, on se heurte inéluctablement au problème des branchements. En effet les instructions de sauts conditionnels sont extrêmement fréquentes dans un programme (souvent plus d'une instruction sur six). Il est intéressant de pouvoir prédir si un branchement va être pris ou si l'exécution va continuer en séquence. L'idée est de faire une supposition sur la branche qui va être prise et de commencer à l'exécuter. Ceci suppose l'existence d'un mécanisme qui permet d'annuler des instructions exécutées par erreur en cas de mauvaise prédiction. Plusieurs techniques sont utilisées ([BYP+91, LW92]) :

On branche toujours : statistiquement il est connu que dans un programme les branchements sont pris dans la majorité des cas. La méthode consiste à simplement supposer que tous les branchements seront pris et les instructions de cette branche sont exécutées spéculativement.

Prédiction à un bit : on construit une table pendant l'exécution du programme qui mémorise pour chaque instruction de saut rencontrée si le branchement a été pris ou non (les bits de poids faible de son adresse constituent un index dans la table). À chaque instruction de saut, on consulte la table : si elle a déjà été rencontrée on suppose que le comportement sera le même que la fois précédente.

Prédiction à deux bits : on construit encore une table de la même façon que précédemment, mais on stocke deux bits qui correspondent intuitivement au « comportement habituel » et au « dernier comportement ». Après chaque branchement on positionne l'indicateur « dernier comportement » ([HP90, Wal91]). La mise à jour du bit « comportement habituel » est faite dans les deux cas suivants :
– si le branchement est pris et qu'il ait aussi été pris la fois précédente ;
– si le branchement n'est pas pris et qu'il n'ait pas non plus été pris la fois précédente.
Tout se passe comme si une habitude est prise quand le même comportement est adopté deux fois de suite. On peut considérer que ces deux bits constituent un automate à quatre états, représenté sur la figure 2.4.

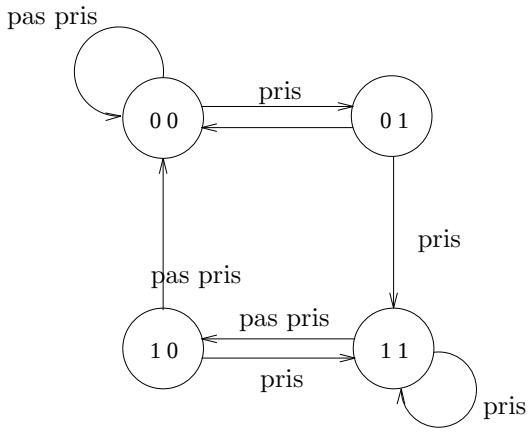

FIG. 2.4 – Automate de prédiction à deux bits

Dans le cas où l'on prédit que le branchement va être pris, il faut déterminer l'adresse de destination. Une méthode consiste à prédire simplement que le processeur va sauter à la même adresse que la dernière fois qu'il a rencontré cette instruction de saut ([Wal91, SJH89]).

Quand des procédures n'ont pas de dépendance de données, il est intéressant de pouvoir les exécuter en parallèle. [TGH92] propose une approche où les sauts dans une procédure n'influent pas sur le séquencement des instructions des autres procédures : la *séparation de procédure*.

2.4.2 Exécution spéculative

Contrairement à la précédente, cette méthode ne choisit pas la branche la plus prometteuse à exécuter lorsqu'un branchement est rencontré : les branches sont exécutées en parallèle.

Cette méthode est la plus efficace : dans la mesure où toutes les branches ont été explorées, il n'y a aucune pénalité de branchement lors de la résolution. Il « suffit » d'annuler les effets des branches prises par erreur.

C'est aussi la plus coûteuse en matériel. Des résultats donnés dans [Uht93] montrent que pour anticiper 32 branchements il faudrait mémoriser $2^{32} \simeq 4 \times 10^9$ chemins, pour un gain de parallélisme réel inférieur à 10.

2.4.3 DEE : Disjoint Eager Execution

La méthode appelée DEE pour *disjoint eager execution*, développée dans [Uht93], est une méthode intermédiaire entre les deux précédentes : prédiction de branchement et exécution spéculative. Elle donne de meilleures performances que la première et requiert moins de matériel que la deuxième. Elle utilise en outre le flux d'instructions statique (c'est-à-dire les instructions telles qu'elles sont en mémoire) qui est indépendant des branchements et permet l'analyse des dépendances de branchements réduites (voir le paragraphe 2.4.4).

Le principe est le suivant : on évalue pour chaque instruction de branchement la probabilité de chacune des branches d'être prise, puis les probabilités cumulées. Dans l'exemple de la figure 2.5 la probabilité de la branche gauche est toujours de 0,7 et celle de la branche droite de 0,3 pour simplifier. Les probabilités des branches diminuent fortement quand le nombre de branches anticipées augmente et finissent par devenir inférieures à celle de la première branche non-prise. C'est cette dernière que l'on va alors explorer. À partir du quatrième branchement, on s'intéresse à des éventualités plus probables ($p = 0,3$) que ne le fait la méthode prédiction de branchement ($p = 0,24$). D'une certaine manière, on se rapproche de l'exécution spéculative : les deux branches possibles sont parcourues et on peut être sûr qu'une partie du travail sera utile puisque l'une des deux branches sera nécessairement prise.

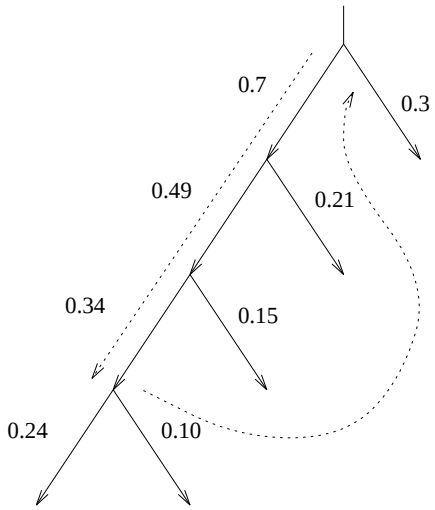

FIG. 2.5 – Évolution de l'exécution DEE

2.4.4 Dépendances réduites

Les méthodes de prédiction de branchement et d'exécution spéculative permettent de réduire le coût des branchements lors de l'exécution mais n'éliminent pas les dépendances inutiles. Pour y remédier, Uht propose dans [Uht91] une analyse détaillée des aléas de contrôle dans le code objet statique. Toutes les dépendances possibles des langages impératifs sont étudiées théoriquement (nids de boucles, boucles disjointes, etc.) et répertoriées. L'ensemble des dépendances est prouvé complet et minimal.

La détection de fausses dépendances est importante dans la mesure où elle permet d'éviter d'appliquer des méthodes de prédictions inutiles et d'extraire du parallélisme supplémentaire. Le gain en performance est important pour un coût matériel modéré.

2.5 Valeurs par adresse

Exécuter plusieurs instructions en parallèle suppose que l'on a pu déterminer auparavant que ces instructions sont indépendantes. Ceci est souvent rendu difficile du fait des indirections et de l'existence de fausses dépendances entre instructions : les valeurs des adresses ne sont souvent calculées qu'au dernier moment, ce qui limite les possibilités d'optimisation.

2.5.1 Fausses dépendances WAR et WAW

On entend par *dépendance* le fait que deux instructions de lecture ou d'écriture ne peuvent être exécutées dans n'importe quel ordre. Il existe *a priori* trois types de dépendances : une lecture après une écriture (*RAW : read after write*), une écriture après une lecture (*WAR : write after read*) et une écriture après une écriture (*WAW : write after write*). Il est clair que deux lectures peuvent être interverties puisque seule l'écriture a un effet destructif.

Les seules vraies dépendances sont les RAW. Ce sont les seules à exprimer une contrainte logique au niveau du code. Les deux autres sont artificielles et produites par le compilateur, elles ne sont présentes en aucune façon dans le source tapé par le programmeur, *i.e.* elles ne font pas partie de la sémantique du programme et il est possible de les éliminer quand elles concernent des registres ([Wal91]). Elles proviennent essentiellement de la manière dont le compilateur travaille : il tend à minimiser le nombre de registres alloués à chaque instant et choisit souvent un nombre limité de registres pour les valeurs temporaires ou les valeurs couramment utilisées dans les boucles ([ASU86, JW89, Wal91]). De fait il a tendance à réutiliser souvent les mêmes registres en créant des dépendances inutiles.

Dans le cas particulier des pipelines, on ne rencontre pas ce type de problème : les données sont lues dans les premiers étages du pipeline (par exemple en DC) et écrites dans les derniers (après WB), comme le montre la figure 2.6. Ceci suffit pour éviter les aléas WAR et WAW. On peut malgré tout se heurter à des difficultés avec des instructions qui ont une latence relativement longue comme les opérations flottantes.

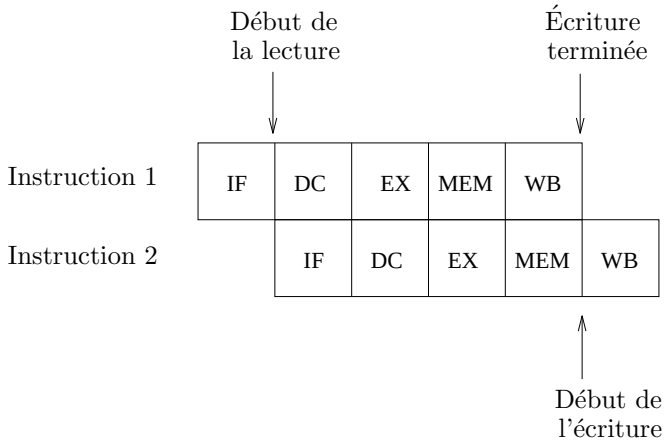

FIG. 2.6 – Absence d'aléas WAR dans les pipelines

Le cas du RAW est résolu par le matériel grâce à la *technique du tableau (scoreboarding)* ([HP90]). Elle consiste en un mécanisme centralisé qui tient à jour une représentation de l'avancement des opérations en cours et de la disponibilité de leurs opérandes, ainsi que l'état des unités fonctionnelles. La détection des aléas et le séquencement des instructions au moment adéquat sont de son ressort. Si une instruction a besoin d'une donnée qui n'est pas encore écrite, la dépendance est détectée et l'instruction est retardée. Ce mécanisme permet l'exécution des instructions dans un ordre différent de l'ordre statique et peut éviter un gel inutile du pipeline.

L'algorithme de Tomasulo ([HP90]) permet aussi de gérer des exécutions parallèles. Le principe ressemble à la technique du tableau avec toutefois deux différences :

- La détection des aléas et le contrôle de l'exécution sont répartis entre les différentes unités fonctionnelles.
- Les résultats sont transmis directement aux unités fonctionnelles via un bus de données commun, sans passer par les registres.

2.5.2 Renommage des registres

Le renommage des registres, voire des adresses mémoires, permet de s'affranchir des fausses dépendances introduites artificiellement par le compilateur. La figure 2.7 en montre un exemple.

Bien que la tendance générale des compilateurs soit d'utiliser un nombre minimum de registres

```
add r1, r2, 3              add r1, r2, 3
ld r2, 5[r6]      ⟶       ld r16, 5[r6]
ld r3, 0[r2]               ld r3, 0[r16]
... r2 ...                 ... r16 ...
```

FIG. 2.7 – Fausse dépendance WAR et résolution

pour stocker les valeurs des calculs intermédiaires (ce qui a pour effet de faire apparaître des fausses dépendances dans le code objet), un compilateur « astucieux » fait attention à l'utilisation qu'il fait des registres.

Une façon d'éviter les aléas de type RAW est d'appliquer la règle d'*affectation unique* : toute variable et tout registre ne doivent être affectés qu'une seule fois. La modification d'une valeur entraîne l'écriture dans une nouvelle cellule mémoire ou dans un registre libre. Ceci peut être fait explicitement pendant l'écriture du programme ou par renommage.

Une méthode simple de renommage est proposée dans [Log72]. Le principe est de commencer par représenter le programme par un automate d'états, puis de définir des *régions* qui correspondent intuitivement aux zones où toutes les occurrences d'une variable représentent effectivement une donnée unique. Toutes les occurrences peuvent être renommées dans cette zone, à condition évidemment que le nouveau nom n'entre pas en conflit avec une autre variable de la région. Le problème du renommage en utilisant un nombre minimum de noms (économie de mémoire ou de registres) est ramené au problème de coloration d'un graphe.

Une variante consiste à confier cette tâche de renommage au matériel ([Wal91]). Celui-ci tient à jour une table d'indirection qui associe à chaque numéro de registre apparaissant dans une instruction le numéro réel du registre utilisé. À chaque fois qu'une instruction modifie la valeur d'un registre, le matériel choisit dynamiquement un registre qui va être utilisé aussi longtemps nécessaire. Le superscalaire IBM RISC 6000 implémente une forme de renommage dans son unité de calcul flottant.

L'allocation dynamique des registres peut donner de meilleurs résultats que l'allocation statique, même dans les cas où le compilateur est particulièrement efficace.

Une technique de réduction de l'occupation des registres dans les boucles connue sous le nom d'*expansion d'une variable modulo* est présentée dans [Bod89]. Le principe est d'allouer des *registres virtuels* à chaque variable, puis de les assigner aux registres de la machine après ordonnancement de la boucle. Plusieurs valeurs différentes d'une même variable peuvent ainsi exister simultanément, ce qui autorise l'exécution en parallèle de plusieurs itérations (cf. le dépliage des boucles et le pipeline logiciel décrits aux paragraphes 2.3.1 et 2.3.2).

Ces formes de renommage s'appliquent difficilement à la mémoire. Toutefois [TGH92] propose quelques pistes pour s'affranchir des fausses dépendances liées à la réutilisation de la pile et pour les machines à flot de données (*machines Dataflow*).

2.5.3 Analyse des alias sur la mémoire

Les fausses dépendances peuvent aussi concerner des références à la mémoire, et ce d'une façon différente des registres. En effet si dans le cas des registres il est facile de détecter quand deux instructions utilisent le même paramètre (le numéro du registre est le même), dans le cas de la mémoire l'analyse se révèle plus complexe. Par exemple dans la séquence de code suivante :

```
ld r1, 0[r9]
st 4[r16], r3
```

il est impossible de savoir si les adresses mémoires référencées sont identiques ou non. Si elles sont différentes, il est possible d'effectuer la lecture et l'écriture dans n'importe quel ordre. Par contre si elles sont égales, on a une dépendance de données entre les deux instructions et il convient

d'être prudent quant à l'ordre d'exécution. Dans le cas où l'on est incapable de déterminer s'il y a dépendance ou non, il est nécessaire de supposer le pire des cas et de perdre du parallélisme potentiel.

L'analyse des alias peut aider un compilateur à décider si deux références sont identiques ou non. On trouve dans [Wal91] une description sommaire de deux techniques possibles :

Analyse d'alias par inspection d'instructions C'est une technique qui consiste à regarder si les deux instructions considérées sont « clairement » indépendantes. Ceci peut se produire dans les deux cas suivants :

```
ld r1, 0[r9]          ld r1, 0[fp]
st 8[r9], r2          st 0[gp], r2
```

Dans le premier cas les deux instructions utilisent le même registre, mais avec des valeurs de déplacement différentes. Dans le deuxième cas une instruction fait référence à la pile tandis que l'autre fait référence à une zone de données. Dans un cas comme dans l'autre, aucun conflit ne peut se produire.

Analyse d'alias par compilateur Un certain nombre de conflits concernant l'accès à des éléments d'un tableau peuvent être résolus grâce à l'analyse du flot de données et à la résolution d'équations diophantiennes[2] : pour déterminer si deux adresses sont égales on écrit l'égalité des expressions qui les définissent et on résout cette équation. Si elle n'a pas de solution on peut être sûr que les adresses sont différentes.

L'utilisation de pointeurs rend cette analyse extrêmement complexe.

2.5.4 Détection des aléas RAW par matériel

Dans de nombreux cas il est impossible de détecter des aléas RAW au moment de la compilation. Le fragment de code suivant donne un exemple :

```
DO I = 1, N
  A(I) = A(B(I)) + 1
ENDDO
```

Il est impossible de savoir si les valeurs de `B(I)` vont créer des dépendances entre les différentes itérations de la boucle et dans le doute on ne peut pas paralléliser. La détection des aléas par matériel permet de résoudre ce type de problème.

Une architecture permettant la mise en œuvre de cette technique est décrite dans [JS86] : après décodage, les instructions sont placées dans les files d'attentes associées aux unités fonctionnelles appropriées. Les adresses des instructions d'écriture en mémoires sont de plus sauvegardées dans un tampon jusqu'à ce que la donnée soit disponible. Les adresses des lectures sont comparées au contenu du tampon et sont exécutées s'il n'y a pas de conflit. Les lectures peuvent ainsi être exécutées tant qu'aucune dépendance n'apparaît, même si une écriture est bloquée en attente de donnée.

2.6 Décodage parallèle

Tjaden et Flynn remarquent dans [TF70] que de nombreux systèmes sont capables d'exécuter plusieurs instructions par cycle et de les décoder à la vitesse maximale de une instruction par cycle, mais qu'aucun ne tente d'en décoder plusieurs simultanément. Les auteurs étudient tout d'abord les différents types de dépendances (fortes/faibles, aléas de données/de contrôle/de structure) et les différentes combinaisons d'instructions possibles, puis proposent une amélioration de l'IBM 7094 et simulent ses performances.

Le processeur possède un jeu de registres à double indice A_i^j, p unités arithmétiques, une unité de contrôle et une pile de prédécodage (*PDS : predecode stack*). Seuls les registres A_i^1 sont

[2] On appelle équation *diophantienne* une équation à coefficients entiers (ou rationnels) dont on cherche les solutions entières.

visibles pour le programmeur, les autres sont utilisés de façon interne au processeur. L'élément inhabituel est le PDS. Son rôle est triple :

1. détecter les instructions fortement indépendantes et transmettre les adresses des instructions faiblement indépendantes pour comparaison ;
2. attribuer les registres A_i^j aux instructions ;
3. maintenir des pointeurs sur les opérandes jusqu'à ce qu'une adresse indépendante soit trouvée.

Le décodage parallèle est particulièrement important pour les processeurs superscalaires. En effet plusieurs instructions sont chargées et le processeur doit les décoder rapidement pour déterminer lesquelles peuvent être exécutées simultanément.

Chapitre 3

Résultats

Cette partie présente les méthodes générales employées pour analyser le parallélisme potentiel existant dans une application, puis les différentes classes de programmes choisies pour les simulations. Enfin les performances de méthodes décrites dans la partie précédente sont brièvement exposées.

3.1 Méthodologie

La principale méthode utilisée pour déterminer quantitativement les performances d'une technique consiste à exécuter un programme de test (*benchmark*) sur une machine « standard » et à obtenir une trace. Celle-ci est alors analysée en fonction des caractéristiques de la machine virtuelle que l'on se donne ([Wal91, LW92, BYP+91, SJH89]). On peut par exemple considérer que l'on dispose d'une machine qui possède un nombre illimité de registres, d'une prédiction de branchement correcte dans 99% des cas ou de plusieurs unités arithmétiques flottantes. Il est possible de simuler le comportement de n'importe quelle machine, pour peu que l'on connaisse ses caractéristiques.

Il faut simuler l'exécution du code objet optimisé et comparer les temps d'exécution séquentiel et parallèle. Le rapport des deux donne le degré de parallélisme. Pour simuler une prédiction de branchement parfaite il est aussi possible d'exécuter une première fois le programme en mémorisant tous les branchements puis une deuxième fois avec le même jeu de données. On connaît alors à l'avance les choix qui seront faits et il est possible de les anticiper.

On peut ainsi évaluer l'influence d'un facteur ou d'un ensemble de facteurs sur les performances globales d'une machine, qu'il s'agisse d'un mécanisme matériel ou d'une optimisation logicielle faite à la compilation. L'intérêt se mesure au rapport entre le parallélisme exploité et le coût d'une telle mise en œuvre (en temps, en complexité et en matériel).

Les différents articles présentent en général un éventail de machines virtuelles avec leurs paramètres respectifs. Elles sont plus ou moins réalistes. Les plus réalistes donnent une idée de ce qui nous est facilement accessible, les plus ambitieuses donnent essentiellement une borne supérieure au parallélisme exploitable, en d'autres termes ce que l'on est sûr de ne jamais pouvoir atteindre.

Au bas de l'échelle on trouve les machines *Base* ou *Stupid* qui ne sont capables d'aucune optimisation et se contentent d'exécuter les instructions une par une, dans l'ordre, sans aucune tentative d'anticipation d'un résultat. Elles correspondent aux ordinateurs « classiques ». À l'autre extrémité existent des modèles comme *Oracle*, *Perfect* ou *Omniscient* qui sont capables de toutes les optimisations considérées avec un taux de réussite maximal (prédiction de branchement parfaite, analyse des alias parfaite, renommage avec une infinité de registres, ...). Toute une gamme de machines s'étend entre ces deux extrêmes (*Myopic, Fair, Speculator*), qui permet d'évaluer l'importance des paramètres les uns par rapport aux autres.

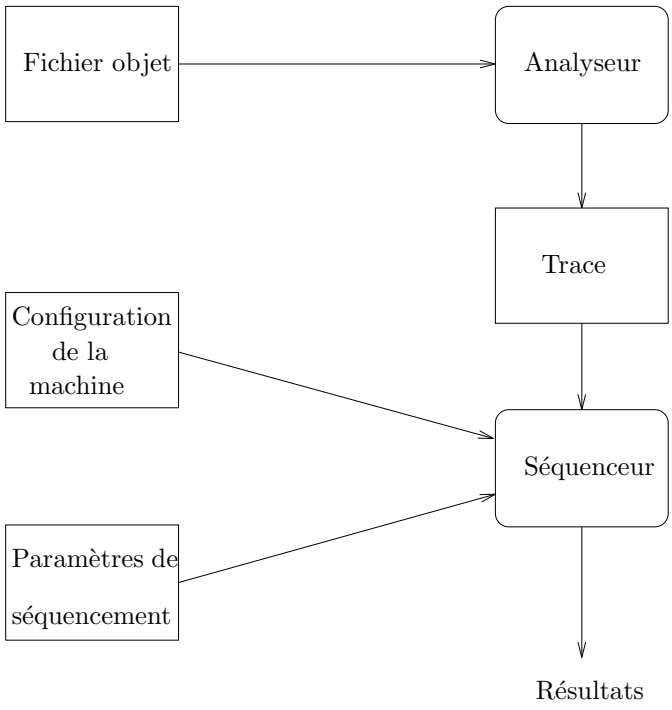

FIG. 3.1 – Simulateur

3.2 Programmes testés

À la lumière des résultats des différents tests, il apparaît que les applications peuvent être classées en trois grandes catégories :

Logiciels de calcul scientifique Ils sont souvent constitués d'une succession de boucles imbriquées, itérées un grand nombre de fois (typique dans le cas d'un produit de matrices, d'une transformée de Fourier, ...).

Benchmarks « classiques » On rencontre aussi les programmes de test comme Linpack, Livermore et Whetstones. Ils ne sont jamais pris seuls pour faire un banc d'essais, mais toujours avec des applications des deux autres catégories.

Autres Il s'agit en général de compilateurs (gcc, Fortran, yacc), de formatteurs de texte (TEX, LATEX), de manipulation de fichiers (grep, diff). Ils ne font quasiment aucun calcul flottant et ne manipulent pas les entiers de façon intensive.

Les applications de calcul scientifique contiennent un degré de parallélisme potentiel particulièrement élevé. Des valeurs de l'ordre de la centaine ou du millier d'instructions par cycle sont détectées ([BYP+91, Wal91, LW92]) grâce à des modèles de machines mettant en œuvre les techniques précédemment exposées.

À l'opposé, il semble communément admis qu'il est difficile d'extraire du parallélisme d'instructions des programmes non-scientifiques. Les valeurs restent en général inférieures à la dizaine, voire inférieures à quelques unités. Les auteurs de [SJH89] se sont intéressés spécifiquement à ce type d'application et concluent qu'il n'est pas trop difficile de trouver deux instructions à exécuter à chaque cycle mais montrent qu'il y a toutefois des obstacles à surmonter.

3.3 Influence de différents paramètres

3.3.1 Régularité du parallélisme

La régularité du parallélisme est une notion qui n'est que très rarement abordée dans la littérature. Elle fournit pourtant une information importante sur la « qualité » du parallélisme d'instructions utilisable. Un exemple permet de mieux comprendre l'influence de ce facteur.

On considère le cas extrême d'un programme constitué d'une partie purement séquentielle (les dépendances peuvent faire en sorte que chaque instruction dépend de la précédente) et d'une partie dont le degré de parallélisme est p. Le programme comporte N instructions qui s'exécutent toutes en un cycle et la proportion du code susceptible d'être parallélisé est notée α. Le parallélisme moyen du programme est :

$$P_{moyen} = \frac{N}{N(1-\alpha) + \frac{\alpha N}{p}} = \frac{p}{\alpha(1-p) + p}$$

Cette valeur est supposée constante et on étudie le taux d'utilisation des P_{moyen} unités fonctionnelles ou processeurs utilisés. Le nombre de cycles requis est $N(1-\alpha) + \frac{\alpha N}{P_{moyen}}$. Le parallélisme apparent est donc :

$$P_{réel} = \frac{N}{N(1-\alpha) + \frac{\alpha N}{P_{moyen}}} = \frac{p}{\alpha^2(1-p) + p}$$

La figure 3.2 montre l'évolution du taux d'utilisation des processeurs en fonction de α pour des degrés de parallélisme théoriques constant égaux à 5, 10 et 20.

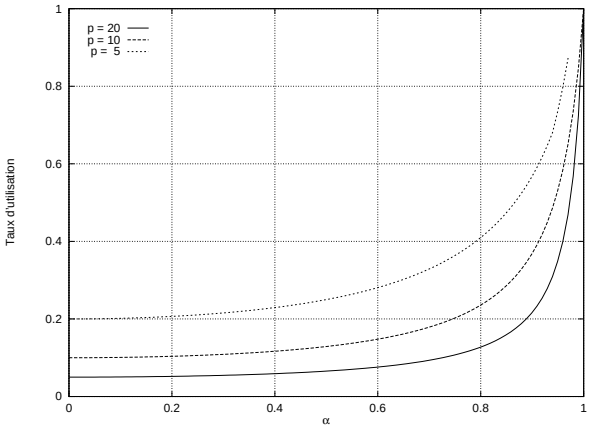

FIG. 3.2 – Régularité du parallélisme

Les résultats des simulations donnés dans [TGH92] montrent que la régularité est supérieure à 85% pour le modèle *Oracle Omniscient* qui dispose du renommage des registres et de la mémoire, d'une prédiction de branchement parfaite et d'une fenêtre d'instructions infinie. Le *Multithreaded Speculator* n'effectue pas d'exécution spéculative et n'atteint que des valeurs comprises entre 55% et 75%.

3.3.2 Optimisation des boucles

Les optimisations de boucles ont un effet sensible sur les performances d'un programme en terme de vitesse d'exécution mais l'effet sur le degré de parallélisme est moins net ([LW92]) : le dépliage d'une boucle provoque deux phénomènes contradictoires.

- En éliminant les branchements et les dépendances entre les indices de boucles, on obtient un code qui se prête bien à une exécution parallèle.
- En supprimant les délais en début de boucle (*overhead*), on supprime par là même des portions de code qui s'exécutaient en parallèle avec d'autres instructions du programme.

La prédominance de l'un de ces deux effets sur l'autre est fonction du programme et de l'ordinateur. L'accroissement du parallélisme est plus important quand d'autres méthodes sont appliquées simultanément (prédiction de branchement ou analyse d'alias). Les programmes dont le flot de contrôle est particulièrement complexe sont généralement indifférents à ce type d'optimisation.

3.3.3 Prédiction de branchement et de saut

L'efficacité des mécanismes matériels de prédiction de branchement est très nette : plus de 90% de bonnes prédictions dans [Wal91, LW92]. Et l'existence de ces mécanismes a un effet important sur le parallélisme comme le montrent les résultats de Wall dans [Wal91] et de Lam et Wilson dans [LW92] (la prédiction de saut a toutefois un effet moindre tant que la prédiction de branchement n'est pas parfaite). En effet les instructions de branchement sont fréquentes et une mauvaise prédiction entraîne l'apparition d'une *bulle* dans le pipeline, c'est-à-dire un série de cycles pendant lesquels aucune exécution n'a lieu, alors que des chargements et des décodages inutiles sont lancés. Les conséquences sur les performances sont sensibles. On trouve dans [Wal91] un graphique qui montre la dégradation du parallélisme en fonction du nombre de cycles vides insérés après chaque mauvaise prédiction.

La conclusion est la même dans [LW92] où les modèles de machines sont légèrement différents. Les auteurs étudient en outre le rapport entre la longueur des blocs d'instructions exécutés sans mauvaise prédiction et le parallélisme du bloc.

3.3.4 Résolution des fausses dépendances

L'analyse des alias permet d'accroître le parallélisme dans certaines mesures. Wall a remarqué que l'analyse par inspection des instructions augmente rarement le degré de parallélisme de plus de 0,5 unités. Par contre l'analyse par compilateur fournit de meilleurs résultats, mais qui restent largement dépendants du programme testé : `sed` obtient un parallélisme de 8 alors que `tomcatv` atteint la valeur de 52 ([Wal91]). Comme dans le cas de l'optimisation des boucles, les améliorations sont plus fortes quand le modèle dispose de la prédiction de branchement et de saut parfaite.

Le renommage des registres permet aussi une nette amélioration dans le cas où tous les autres paramètres de la simulation sont considérés comme parfaits. Le gain est beaucoup plus faible dès que l'on s'intéresse à des modèles moins ambitieux comme la machine *Great* de [LW92] (qui reste pourtant utopique). On peut remarquer qu'il y a peu de différences entre un modèle avec 256 registres et un modèle qui autorise un renommage avec un nombre de registres infini.

Deuxième partie

Rapport de stage

Introduction

Le besoin constant de puissance de calcul a conduit à développer de nombreuses techniques, tant logicielles que matérielles, pour accroître le degré de parallélisme d'un programme. Nombre d'hypothèses sont faites sur les caractéristiques des ordinateurs utilisés comme modèles dans les simulations et certaines sont loin d'être réalistes. En conséquence les valeurs obtenues ne doivent être comprises que comme des bornes supérieures du parallélisme qui nous est accessible.

Il est maintenant intéressant de voir dans quelle mesure ce parallélisme peut être mis en œuvre. En effet le choix d'une architecture — nombre d'unités arithmétiques, d'unités flottantes, profondeur et structure du pipeline, etc. — est une phase importante dans la conception d'un processeur et il doit passer par l'étude du comportement des logiciels en fonction du matériel. Il est notamment essentiel de comprendre pourquoi certaines architectures n'apportent que de piètres performances et comment les pertes se sont manifestées pour tenter d'apporter une amélioration significative. Il est aussi intéressant de voir comment les familles de logiciels utilisés (calcul scientifique, compilation, manipulation de textes, ...) influent sur le parallélisme exploitable, indépendamment du matériel.

La simulation apparaît comme un outil idéal pour tester facilement et rapidement de nombreuses possibilités. Le premier chapitre montre la complexité supplémentaire apportée par l'aspect superscalaire des processeurs. Il rappelle les particularités de ce type de processeurs et caractérise plus précisément les aléas susceptibles d'intervenir au cours de l'exécution d'un programme. Le fonctionnement et les principes du simulateur sont exposés au chapitre 2 qui développe l'ensemble des possibilités du simulateur et des informations qu'il est possible de collecter. Le chapitre 3 présente les architectures particulières auxquelles on s'est interessé dans cette étude avec les cas particuliers qu'elles introduisent ainsi que les programmes utilisés pour les tests. Les résultats et conclusions figurent au chapitre 4.

Chapitre 4

Pourquoi simuler un processeur superscalaire ?

4.1 Particularités d'un processeur superscalaire

Un processeur supercalaire se distingue essentiellement d'un processeur scalaire par le fait qu'il est capable d'exécuter simultanément plusieurs instructions. Cette différence entraîne un surcroît de complexité matérielle et fait apparaître nombre de situations de blocages qui n'existent pas dans un processeur scalaire. Il est impossible d'extrapoler le comportement d'un tel processeur à partir de données sur un processeur scalaire.

On va s'intéresser ici à des processeurs superscalaires « naturels » et « symétriques ». Les premiers ne contiennent qu'un seul exemplaire de chaque unité fonctionnelle, toutes pouvant être utilisées simultanément. Un processeur superscalaire est dit « symétrique » dès lors que certaines de ses unités fonctionnelles, considérées comme les plus importantes, sont dupliquées.

4.2 Pertes de performances

Les performances atteintes sont loin d'être égales au maximum théorique qui est d'une instruction par cycle et par pipeline, soit cinq instructions par cycle avec un processeur superscalaire de degré cinq. Cette perte s'explique par le fait que de nombreuses instructions restent bloquées en attente de la fin d'une instruction précédente. Comme on l'a vu dans la première partie, ces pertes peuvent être classées en trois catégories :

Conflits de ressources : une unité fonctionnelle requise est déjà utilisée par une instruction précédente. Cet aléa est lié directement à l'architecture du processeur. Il pourrait être résolu par l'ajout d'une nouvelle unité fonctionnelle.

Aléa de données : deux cas distincts peuvent se présenter :

- Un résultat d'une instruction précédente nécessaire n'est pas encore disponible. Il s'agit d'une lecture après écriture (ou RAW).
- Une instruction va écrire son résultat alors que la précédente doit écrire au même endroit et ne l'a pas encore fait. On parle d'écriture après écriture (ou WAW).

Délais de branchement : après une instruction de branchement, l'adresse de la cible n'est pas connue immédiatement. On ne sait donc pas quelle instruction il faut charger.

Ces blocages affectent fortement le parallélisme des programmes. Ils ont, par exemple, fait chuter le degré de parallélisme de tous les programmes que nous avons simulés sur un processeur superscalaire de degré cinq à une valeur inférieure à deux.

4.3 Classification des aléas

Sur un processeur scalaire pipeliné, plusieurs dépendances de données peuvent bloquer une instruction : le délai de chargement (*load delay*) (cf. figure 4.1) peut se produire si une instruction

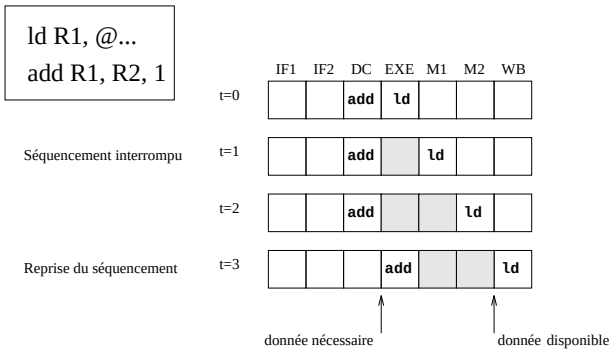

FIG. 4.1 – Délai de chargement

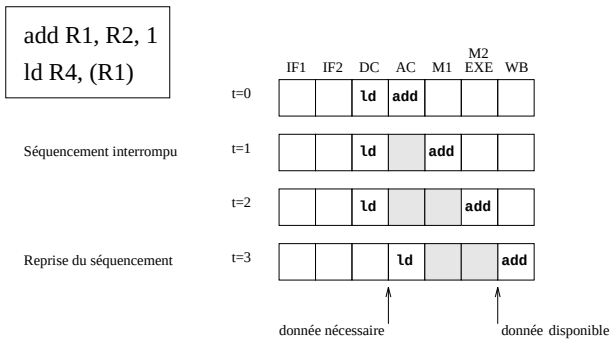

FIG. 4.2 – Délai d'adressage

de lecture en mémoire demande un certain nombre de cycles avant de pouvoir fournir un résultat. Dans les cas où l'instruction suivante a besoin de cette valeur, elle doit attendre. Quand l'étage d'exécution est repoussé plus tard dans le pipeline, deux autres cas entrent en jeu : le délai d'adressage (*address delay*) et le délai d'écriture (*store delay*). Le premier correspond au cas où une instruction qui doit accéder à la mémoire ne peut effectuer son calcul d'adresse parce qu'il dépend, par exemple, du résultat d'une addition précédente (cf. figure 4.2) ; le deuxième correspond au cas d'une instruction d'écriture (`store`) qui ne dispose pas encore de la valeur à écrire (cf. figure 4.3). Il ne se produit que lorsque le nombre d'étages de chargement est supérieur ou égal à deux.

Dès lors que l'on autorise plusieurs instructions à s'exécuter simultanément, le nombre de conflits augmente et il devient parfois difficile de classer certaines situations dans une catégorie précise. Pour permettre une analyse plus fine du comportement des programmes et de l'importance relative des différents facteurs, une classification plus détaillée des aléas de données s'est avérée nécessaire [1].

4.3.1 Aléas RAW

Les différentes configurations de pipelines qui existent conduisent à des situations de blocages diverses. Toutes les situations possibles sont recensées dans la liste ci-dessous avec un exemple. Les tableaux 6.1, 6.2 et 6.3 indiquent les classes d'instructions entre lesquelles les aléas peuvent se produire pour les pipelines qui seront considérés dans la suite de cette étude. Le paragraphe 6.2 apporte de plus amples précisions.

1. **chargement entier :** une instruction entière tente de faire un calcul et utilise le résultat d'un `load` précédent qui ne s'est pas encore terminé. La donnée n'est pas disponible immédiatement.

[1] Les aléas de type WAR n'interviennent pas dans les architectures pipelines que nous avons considérées dans ce rapport.

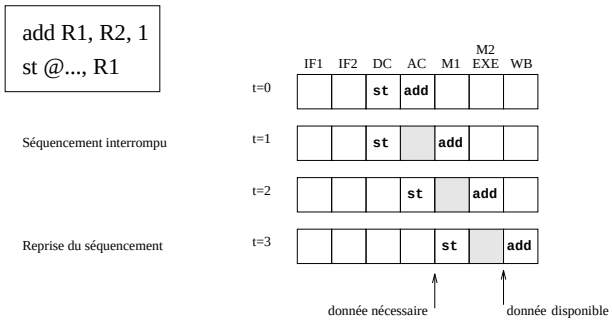

FIG. 4.3 – Délai d'écriture

```
ld  R1 ← 0[R2]
add R3 ←  R1, R4
```

2. **chargement flottant :** idem avec un calcul flottant.

```
ldf  F1 ← [R2]
fadd F3 ←  F1, F4
```

3. **adressage :** l'adresse à laquelle une donnée doit être lue (`load`) ou écrite (`store`) est calculée par une instruction précédente. L'étage de calcul d'adresse ne peut commencer.

```
add  R1 ← R2, R3
ld R4 ← 4[R1]
```

4. **écriture entier :** l'écriture d'une donnée entière ne peut commencer car la valeur ne sera pas fournie à temps par une instruction précédente.

```
add  R1 ← R2, R3
st 2[R4] ←  R1
```

5. **écriture flottant :** idem avec une donnée flottante.

```
fadd  F1 ← F2, F3
stf 2[R4] ←  F1
```

6. **RAW flottant :** une instruction flottante utilise le résultat de la précédente et ne peut commencer tant que l'autre n'a pas terminé (les instructions flottantes nécessitent typiquement plusieurs cycles).

```
fadd  F1 ← F2, F3
fadd F4 ←  F1, F5.
```

7. **chargement d'adresse :** l'adresse utilisée par une instruction de lecture ou d'écriture en mémoire utilise le résultat d'un `load` précédent.

```
ld  R1 ← 0[R2]
st 9[R1] ← R4
```

8. **écriture de la donnée chargée :** c'est la donnée à écrire en mémoire qui provient d'un `load`.

```
ld  R1 ← 5[R2]
st 0[R3] ←  R1
```

9. **RAW entier :** il apparaît si le processeur contient plus d'une unité entière et que deux instructions entières dépendantes l'une de l'autre tentent d'être séquencées dans le même groupe.

```
add  R1 ← R2, R3
add R4 ←  R1, R5
```

Ces différents types d'aléas sont ensuite regroupés dans les catégories définies précédemment qui sont les délai de chargement, délai d'adressage, délai d'écriture. Ce regroupement dépend de la structure de pipeline étudiée (cf. paragraphe 6.2).

4.3.2 Aléas WAW

De même les dépendances de sorties peuvent être réparties en cinq sous-ensembles. Il peut là aussi y avoir des distinctions selon que la structure de pipeline à laquelle on a affaire.

1. **entier-chargement :** une opération entière tente d'écrire alors que le `ld` précédent ne l'a pas encore fait.

```
ld  R1 ← 0[R4]
add  R1 ← R2, R3
```

2. **flottant-chargement :** une opération flottante tente d'écrire alors que le `ld` précédent ne l'a pas encore fait.

```
ldf  F1 ← 0[R4]
fadd  F1 ← F2, F3
```

3. **chargement-entier :** un `ld` tente d'écrire alors que l'opération entière précédente n'est pas encore finie.

```
add  R1 ← R2, R3
ld  R1 ← 0[R4]
```

4. **chargement-flottant :** un `ld` tente d'écrire alors que l'opération flottante précédente n'est pas encore finie.

```
fadd  F1 ← F2, F3
ldf  F1 ← 0[R4]
```

5. **WAW entier :** deux instructions tentent d'écrire dans le même registre entier. Ceci est possible si un branchement est séquencé en même temps qu'une instruction entière (`call` et `jmpl` modifient un registre entier) ou si deux instructions entières sont séquencées en même temps.

```
add R1 ← R2, R3
add R1 ← R4, R5
```

Les tableaux 6.4, 6.5 et 6.6 montrent les séquences d'instructions qui conduisent à chaque dépendance selon le pipeline choisi.

4.3.3 Délais de branchement

Environ 15 % des instructions d'un programme sont des instructions de branchement. Elles sont à l'origine d'une sérieuse dégradation des performances puisque le flot de contrôle est interrompu toutes les six instructions en moyenne. Le chargement ne peut être repris que lorsque la cible est connue. Une prédiction de branchement peut être mise en œuvre pour tenter de prédire l'adresse cible et atténuer ainsi les pertes.

4.3.4 Conflits de ressources

Les conflits de ressources se produisent dans un processeur superscalaire. Le processeur est capable d'exécuter un certain nombre d'instructions simultanément, mais toutes les combinaisons ne sont pas forcément possibles. Par exemple si deux unités flottantes sont prévues et qu'une séquence d'un programme contienne trois instructions flottantes indépendantes successives, la troisième devra attendre un cycle.

Chapitre 5

Comment ?

5.1 Chaîne d'analyse

La simulation proprement dite d'un programme avec une configuration matérielle donnée est précédée d'un certain nombre d'étapes qui permettent d'aboutir à un résultat correct. La figure 5.1 illustre cette chaîne d'analyse.

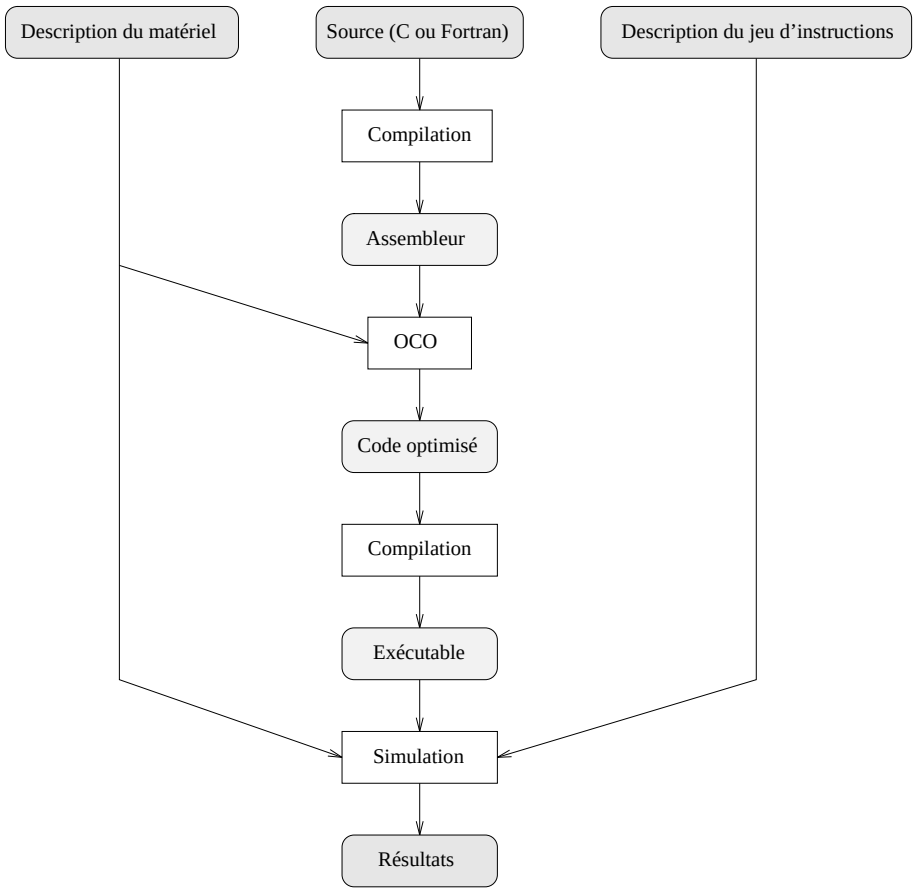

FIG. 5.1 – Chaîne d'analyse

- La chaîne commence par la génération à partir du source (en C ou en Fortran) d'un code en langage d'assemblage. Il est essentiel d'activer toutes les options d'optimisation (`cc -O4` ou `f77 -O4`) pour obtenir des résultats significatifs.
- Les programmes simulés nécessitent d'être optimisés en fonction de l'architecture avec laquelle ils vont être simulés. Cette optimisation consiste à tenir compte des « bulles » qui vont apparaître dans les pipelines (c'est-à-dire des pertes de cycles dues aux interblocages) et à tenter d'insérer des instructions utiles pour combler les pertes. Cette optimisation est

réalisée sur le code source en assembleur produit par le compilateur. Le logiciel *Oco* (*object code optimizer*), développé à l'Irisa, est capable de réordonnancer le code au niveau des blocs de base [1] à partir d'une description du jeu d'instructions de la machine sous forme de tables de réservations. Plus précisément : *Oco* construit un graphe de dépendances basé sur l'utilisation des registres pour chaque bloc de base. *Oco* ne fait que déplacer des instructions les unes par rapport aux autres. Il n'ajoute aucune instruction (à l'exception de NOP) et ne modifie pas les numéros des registres.
Le code est réordonnancé pour chaque configuration.

– Le programme recompilé est exécuté avec *Spy*, un outil développé par Gordon Irlam ([Irl91]), qui exécute un programme sur une station de travail Sun Sparc et récupère les codes opératoires ainsi que les adresses des instructions et des données. Ces informations sont ensuite passées au simulateur qui sait les exploiter.

5.2 Simulateur

5.2.1 Principes

Après une phase d'initialisation pendant laquelle il lit des fichiers de description de l'architecture à simuler, le simulateur reçoit en entrée les codes opératoires des instructions du programme un à un, ainsi que les adresses des données et des instructions. Les instructions ne sont pas réellement exécutées. Seules les dépendances qu'elles vont engendrer sont prises en compte.

Le simulateur travaille cycle à cycle : il reçoit les instructions une à une et les place dans l'étage de chargement du pipeline. On a supposé ici qu'il est toujours possible de remplir cet étage, c'est-à-dire qu'un processeur superscalaire de degré n peut lire n mots dans le cache d'instructions, même s'ils sont dans deux lignes différentes [2]. On suppose en outre que l'on dispose d'un bus de données et d'un bus d'instructions et que la lecture d'une donnée ne gêne pas le chargement des instructions au même cycle.

Les instructions sont exécutées dans l'ordre où elles sont reçues. Le simulateur ne modifie en rien l'ordre des instructions. On suppose que toutes les optimisations utiles ont dû être faites par logiciel auparavant. Le pipeline est synchrone : les dépendances sont testées au moment du séquencement. Une instruction séquencée est donc sure que les unités fonctionnelles dont elle aura besoin seront disponibles au moment voulu et que ses données seront présentes. Les instructions sont séquencées par groupes qui se propagent dans le pipeline. Ceci permet d'éviter le phénomène de *bubble squashing* [3], difficile à gérer tant au niveau logiciel qu'au niveau matériel, et simplifie grandement le comptage et l'analyse des interblocages. Les court-circuits (ou *bypass*) sont aussi simulés : une instruction n'est pas obligée d'attendre qu'une valeur soit écrite dans le fichier de registres pour pouvoir l'utiliser si elle est présente sur le bus de données.

Deux caches associatifs, un pour les instructions et un pour les données, sont adjoints au processeur. Ils sont considérées comme parfaits au niveau de la simulation : toutes les valeurs accédées sont présentes dans le cache. Le nombre de défauts est toutefois comptabilisé pour les instructions comme pour les données. Ceci permet d'évaluer ensuite séparément l'impact de la hiérachie mémoire sur les performances du processeur.

Un algorithme de prédiction de branchement peut aussi être mis en œuvre pour améliorer les performances. L'algorithme retenu utilise une table de prédiction à deux bits associée à un buffer de prédiction de branchement (*branch target buffer* ou *BTB*). Ces deux éléments se comportent comme des mémoires caches. Étant donné l'adresse d'un branchement, la table indique s'il est préférable de suivre la branche prise ou non et le BTB donne l'adresse cible prédite. Si l'on a choisi des caches à accès direct, les informations peuvent être obtenues pendant le premier étage

[1] Un bloc de base est une séquence d'instructions qui ne contient pas de branchement et qui est délimitée par des branchements.

[2] Ceci pose des problèmes au niveau de la réalisation matérielle du cache puisque deux ports sont nécessaires.

[3] C'est-à-dire le fait qu'une instruction séquencée puisse être bloquée et rattrapée par la suivante : la bulle qui s'était formée entre les deux est écrasée.

de chargement : la pénalité résultante est nulle en cas de bonne prédiction, elle est égale au nombre d'étages placés avant l'étage d'exécution en cas de mauvaise prédiction. Le simulateur utilise des tables à 256 entrées.

5.2.2 Fonctionnement

Le fonctionnemement du simulateur est illustré sur la figure 5.2. Le code source du paquetage

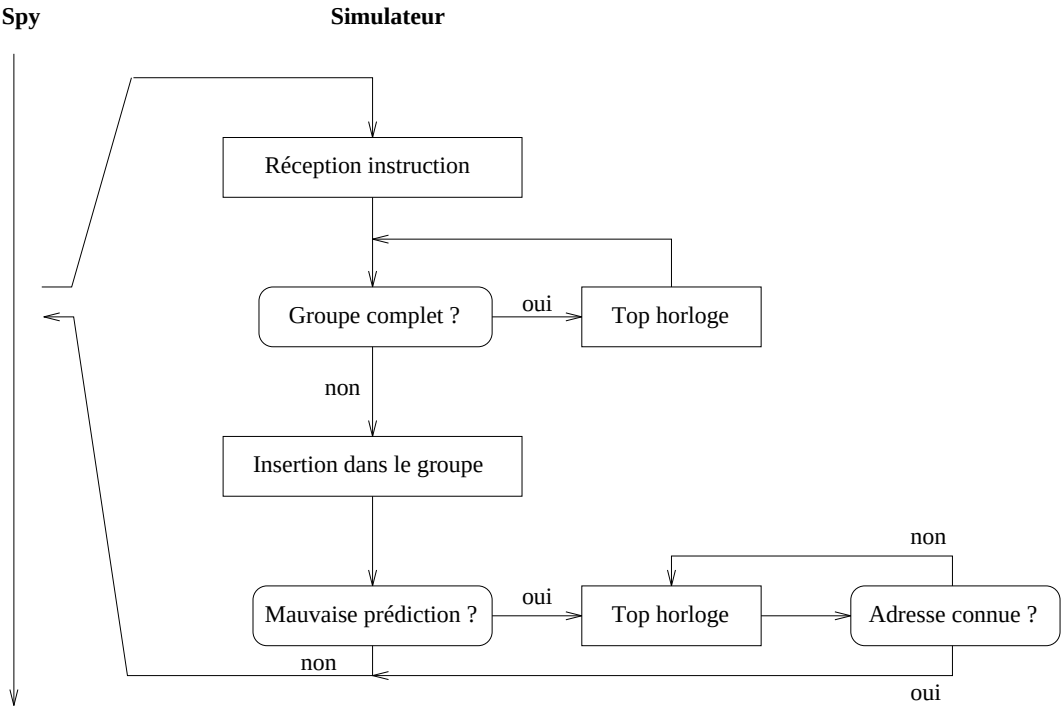

FIG. 5.2 – Fonctionnement du simulateur

Spa est disponible et il est facile de rajouter un appel de fonction pour obtenir les informations nécessaires au simulateur. Les instructions sont placées l'une après l'autre dans ce qui correspond au premier étage de chargement du pipeline (IF1). Lorsque le groupe est complet, un top d'horloge est simulé. En cas de mauvaise prédiction de branchement il faut attendre que l'instruction de saut ait terminé son exécution pour connaître l'adresse cible et pouvoir recommencer à charger des instructions du cache.

La simulation d'un « top d'horloge » effectue un certain nombre d'opérations :

1. toutes les instructions déjà séquencées avancent d'un étage dans le pipeline et celle qui en étaient au dernier étage sont supprimées,
2. les instructions suivantes sont testées dans l'ordre de leur chargement pour déterminer si elles peuvent être séquencées,
3. les dépendances et les réservations d'unités fonctionnelles sont mises à jour en fonction des modifications précédentes,
4. toutes les autres instructions (en cours de chargement ou de décodage) avancent d'un étage si c'est possible.

Cette boucle se répète jusqu'à ce que le nombre d'instructions voulu soit atteint ou que le programme se termine.

5.3 Possibilités du simulateur

5.3.1 Architectures

Le simulateur est paramétrable. Deux fichiers permettent de décrire finement le processeur que l'on souhaite tester. Un fichier permet d'indiquer la liste des unités fonctionnelles présentes dans le processeur ainsi que leur nombre (par exemple : deux unités entières, une unité flottante et une unité d'accès à la mémoire). Il précise le nombre maximum d'instructions qui peuvent être séquencées à chaque cycle, c'est-à-dire une pour un processeur scalaire ou le degré pour un processeur superscalaire.

On peut indiquer la mise en œuvre d'un algorithme de prédiction de branchement et le nombre de cycles perdus en cas de mauvaise prédiction (ou s'il n'y a pas de prédiction du tout).

Chaque instruction doit ensuite être détaillée précisément par une table de réservations qui indique quelles unités fonctionnelles sont nécessaires à l'exécution complète de l'instruction et à quel cycle. Elle doit préciser aussi à quel moment les registres source et destination sont lus et/ou écrits.

5.3.2 Informations collectées

Des informations concernant le programme simulé sont collectées à deux niveaux. Tout d'abord les caractéristiques du programme lui-même sont recueillies : nombre d'instructions entières, flottantes, nombre de lectures et d'écritures en mémoire et nombre de branchements. Ensuite viennent les résultats de la simulation proprement dite :

- nombre de cycles nécessaires à l'exécution du programme et nombre moyen d'instructions par cycle (IPC),
- efficacité de la prédiction de branchemement le cas échéant : taux de bonnes prédictions et de mauvaises prédictions,
- nombre de défauts de cache pour les données et les instructions,
- nombre d'« occasions perdues » à cause des délais de chargement, délais d'adressage, délais d'écriture et délais de branchement. Les occasions perdues constituent un moyen de mesurer les pertes de performances dues aux dépendances. En effet le décompte du nombre de cycles perdus n'est représentatif que dans le cas d'un processeur scalaire. Le nombre d'occasions permet de tenir compte du fait que le blocage d'une instruction n'a pas le même impact selon sa position dans le groupe. La figure 5.3 illustre le phénomène pour un processeur de degré 4 avec deux étages de chargement. Le terme « occasion » est utilisé pour

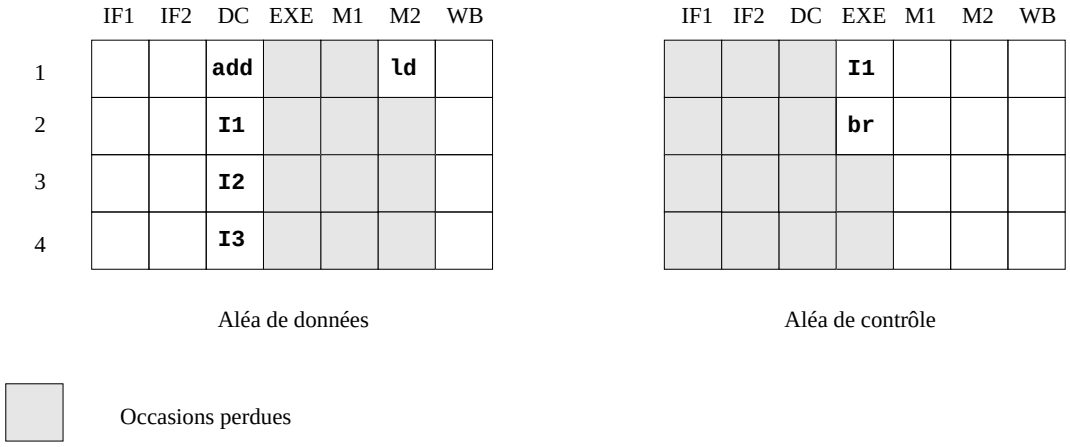

FIG. 5.3 – Occasions perdues

rappeler que la perte entrainée par le conflit n'est pas nécessairement aussi importante. Un autre conflit, dû à l'instruction suivante, aurait pu empêcher le remplissage complet des pipelines.

Le nombre d'occasions perdues liées à un aléa de données est donné par la formule

$$N_{pertes} = d \times n_{load} + (d - pos + 1)$$

où d est le degré, n_{load} le nombre d'étages de chargement et pos la position qu'aurait eu l'instruction dans le groupe si elle avait été séquencée.
Le nombre d'occasions perdues à cause d'une mauvaise prédiction de branchement est

$$N_{pertes} = d \times et_{EXE} - pos$$

où et_{EXE} est la position de l'étage d'exécution dans le pipeline.

– détail des pertes : un récapitulatif de tous les interblocages qui ont eu lieu pendant l'exécution du programme qui indique la raison du gel du pipeline ainsi que le nombre de cycles nécessaire à la résolution du conflit et le nombre d'occasions perdues à chaque fois.

Certains interblocages peuvent avoir plusieurs raisons. Par exemple l'extrait de programme suivant exécuté sur un processeur superscalaire doté d'une seule unité flottante va provoquer un gel du pipeline.

```
fadd F1  ←  F2, F3
fadd F4  ←  F1, 1
```

On peut considérer que le deuxième `fadd` n'a pu être séquencé parce que l'unité flottante est déjà occupée (conflit de ressource) ou qu'il s'agit d'une dépendance de données sur le registre `F1` dont la valeur n'est pas connue avant plusieurs cycles.

Par convention les occasions perdues sont donc comptées avec l'ordre de priorité suivant :

1. délai de branchement,
2. conflit de ressource,
3. dépendance de données : RAW puis WAW.

Chapitre 6

Applications

6.1 Structures de pipelines étudiées

Cette étude avait pour but de comparer les performances qu'il est possible d'atteindre grâce à deux structures de pipelines qui diffèrent par la position de l'étage d'exécution (cf. figure 6.1) : le pipeline GE (pour *greedy execution*) et le pipeline DACS (pour *dedicated address computation stage*).

Une troisième structure, le pipeline Mixte, a été ajoutée au vu des premiers résultats.

Pipeline GE : c'est le pipeline qui est souvent choisi pour les processeurs récents (DECalpha, MIPS R3000/R4000, Intel i860)(cf. [SKV93, LS92]). Les étages d'exécution et d'accès à la mémoire sont distincts. L'étage d'exécution est placé le plus tôt possible, c'est-à-dire immédiatement après le décodage. Les résultats des opérations entières sont donc aussi disponibles au plus tôt. Ceci permet de calculer l'adresse d'une donnée par une instruction entière placée immédiatement avant un `load` ou un `store`. Le mécanisme de court-circuit (*bypass*) permet d'éviter toute perte de cycle.

Par contre une instruction qui utilise le registre destination d'une instruction de chargement précédente est pénalisée : elle reste bloquée au moins un cycle, victime d'un délai de chargement.

Pipeline DACS : comme son nom l'indique, ce pipeline possède un étage spécifique au calcul d'adresse. Il regroupe l'étage d'exécution avec le dernier étage d'accès à la mémoire. Ceci permet d'éviter les délais de chargement en plaçant l'étage d'exécution le plus tard possible. L'idée est la suivante : les codes numériques contiennent souvent une séquence d'instruction du type *lecture d'une donnée-utilisation de cette donnée* qu'il est intéressant de favoriser. Cette solution a été retenue dans le MIPS TFP et dans le Pentium d'Intel.

Dans ce cas, l'instruction qui suit la lecture d'une donnée en mémoire n'est pas pénalisée : la donnée est disponible sur le bus au moment où elle est nécessaire. En contrepartie, les lectures et écritures en mémoire sont souvent défavorisées car elles utilisent comme adresse le résultat d'une opération entière qui n'est pas disponible immédiatement.

D'un point de vue matériel, ce pipeline requiert un additionneur supplémentaire dédié au calcul d'adresse à l'étage AC.

Ces deux structures de pipelines favorisent chacunes certaines séquences d'instructions au détriment d'autres.

Les premiers résultats des expérimentations tendaient à indiquer que le pipeline DACS avait un mauvais comportement sur les programmes qui ne font que du calcul entier, tandis que la différence était plus faible pour les programmes de calcul flottant. L'intuition a été la suivante : le pipeline DACS est mieux adapté aux instructions flottantes, mais toute la partie « contrôle » des programmes introduit une dégradation des performances dans les programmes flottants. Ceci conduit à imaginer une troisième structure de pipeline, compromis entre les deux premières.

Pipeline Mixte : les instructions entières et les accès à la mémoire se font au plus tôt selon la méthode GE, alors que les instructions flottantes se font au plus tard, « à la DACS ». Ce

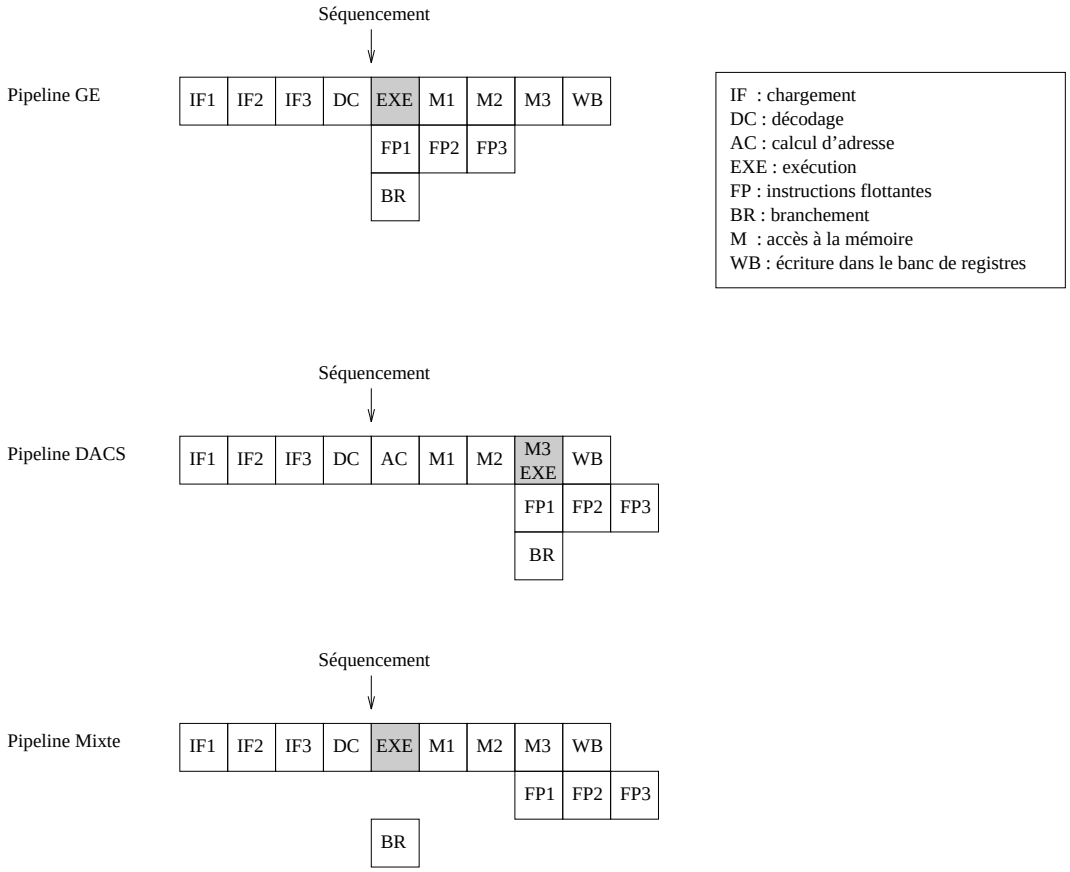

FIG. 6.1 – Pipelines étudiés

pipeline ressemble à celui du SuperSparc (cf. [SKV93]).

Le but des simulations est de déterminer si l'un de ces choix est meilleur qu'un autre et donne les meilleurs résultats dans tous les cas de figures ou si certaines options ont un effet variable selon les programmes.

6.2 Aléas

Le type et la fréquence des aléas diffèrent selon le type de pipeline utilisé. Un recensement précis des circonstances dans lesquelles un aléas peut se produire est résumé dans les tableaux 6.1, 6.2 et 6.3 pour les vraies dépendances et 6.4, 6.5 et 6.6 pour les dépendances de sortie. En abscisse se trouve la première instruction séquencée et en ordonnée celle qui provoque la rupture de séquencement. Les cases remplies en *italique* indiquent que la situation n'est possible que pour un processeur superscalaire dans le cas où les deux instructions étaient candidates pour être séquencées dans le même groupe, les aléas en **gras** ne se produisent que dans un processeur superscalaire qui dispose de plusieurs unités entières.

Ces aléas sont ensuite regroupés sous les dénominations délai de branchement, délai d'écriture et délai d'adressage.

Certains délais peuvent être interprêtés de plusieurs façons différentes. Par exemple le délai de chargement d'adresse peut être vu soit comme un délai de chargement puisque qu'il faut attendre une donnée fournie par un `load`, soit comme un délai d'adressage puisque c'est un calcul d'adresse qui est à l'origine du blocage. Or l'idée à l'origine du pipeline DACS est de retarder au maximum la phase d'exécution pour limiter les délais de chargement, tandis que le pipeline GE favorise les calculs d'adresse. Il a donc été convenu de classer les aléas différemment selon la structure du pipeline pour tenir compte des différents choix. Les classifications sont les suivantes :

pipeline GE
- délai de chargement
  - chargement entier
  - chargement flottant
  - chargement d'adresse
  - écriture de la donnée chargée
- délai d'adressage : adressage
- délai d'écriture : écriture flottant

pipeline DACS
- délai de chargement
  - chargement entier
  - chargement flottant
- délai d'adressage
  - adressage
  - chargement d'adresse
- délai d'écriture
  - écriture entier
  - écriture flottant
  - écriture de la donnée chargée

pipeline Mixte
- délai de chargement
  - chargement entier
  - chargement flottant
  - chargement d'adresse
  - écriture de la donnée chargée
- délai d'adressage : adressage
- délai d'écriture : écriture flottant

Les pipelines GE et Mixte diffèrent uniquement par la position des étages dédiés aux instructions flottantes. C'est la raison pour laquelle ils produisent les mêmes aléas.

	lecture	entier	branchement	flottant
lecture	chargement d'adresse	*adressage*	*adressage*	
écriture (adresse)	chargement d'adresse	*adressage*	*adressage*	
écriture (donnée)	écrit. donnée chargée			écriture flottant
entier	chargement entier	**RAW entier**	*RAW entier*	
branchement	chargement entier	*RAW entier*		RAW entier
flottant	chargement flottant			RAW flottant

TAB. 6.1 – Aléas RAW dans un pipeline GE

	lecture	entier	branchement	flottant
lecture	chargement d'adresse	adressage	adressage	
écriture (adresse)	chargement d'adresse	adressage	adressage	
écriture (donnée)	écrit. donnée chargée	écriture entier	écriture entier	écriture flottant
entier	*chargement entier*	**RAW entier**	RAW entier	
branchement	*chargement entier*	*RAW entier*		RAW entier
flottant	*chargement flottant*			RAW flottant

TAB. 6.2 – Aléas RAW dans un pipeline DACS

6.3 Configurations simulées

6.3.1 Configuration matérielle

Plusieurs architectures différentes ont été simulées avec l'ensemble des programmes de test. À chaque fois quatre temps d'accès à la mémoire cache ont été envisagés (de 1 à 4 étages de chargement), avec et sans prédiction de branchement. Les trois structures de pipeline sont simulées à chaque fois.

	lecture	entier	branchement	flottant
lecture	chargement d'adresse	*adressage*	*adressage*	
écriture (adresse)	chargement d'adresse	*adressage*	*adressage*	
écriture (donnée)	écrit. donnée chargée			écriture flottant
entier	chargement entier	**RAW entier**	*RAW entier*	
branchement	chargement entier	*RAW entier*		RAW entier
flottant	chargement flottant			RAW flottant

TAB. 6.3 – Aléas RAW dans un pipeline Mixte

	lecture	entier	branchement	flottant
lecture		*charg.-entier*	*charg.-entier*	charg.-flottant
entier	*entier-charg.*	**WAW entier**	*WAW entier*	
branchement	*entier-charg.*	*WAW entier*		
flottant	flottant-charg.			

TAB. 6.4 – Aléas de type WAW dans un pipeline GE

	lecture	entier	branchement	flottant
lecture		*charg.-entier*	*charg.-entier*	charg.-flottant
entier	*entier-charg.*	**WAW entier**	*WAW entier*	
branchement	*entier-charg.*	*WAW entier*		
flottant				

TAB. 6.5 – Aléas de type WAW dans un pipeline DACS

	lecture	entier	branchement	flottant
lecture		*charg.-entier*	*charg.-entier*	charg.-flottant
entier	*entier-charg.*	**WAW entier**	*WAW entier*	
branchement	*entier-charg.*	*WAW entier*		
flottant	flottant-charg.			

TAB. 6.6 – Aléas de type WAW dans un pipeline Mixte

Pour des raisons de simplification, on a supposé que toutes les opérations flottantes sont terminées en trois cycles et peuvent être pipelinées.

1. Processeur scalaire simple
2. Processeur superscalaire de degré 4, muni d'une unité entière, d'une unité flottante, d'une unité de branchement et d'une unité d'accès à la mémoire qui peuvent fonctionner simultanément.
3. Processeur superscalaire de degré 5 : le même que précédemment mais doté de deux unités entières.

6.3.2 Programmes simulés

Les programmes utilisés pour les simulations sont au nombre de neuf : cinq de calcul flottant et quatre de calcul entier. Les huit premiers sont issus de la collection SPEC.

Flottant

- mdljdp2 : calcul du mouvement de molécules en interaction,
- wave5 : résolution d'équations de Maxwell,
- tomcatv : génération de maillage,
- alvinn : apprentissage d'un réseau de neurones,
- swm256 : calcul de différences finies ;

Entier

- eqntott : calcul de la table de vérité d'une expression booléenne,
- compress : l'utilitaire de compression de fichiers de Unix,
- xlisp : problème du positionnement de n reines sur un échiquier en Lisp,
- cache : simulation de cache associatifs brouillés.

Tous les programmes ont été réordonnancés avec *Oco* en fonction de l'architecture choisie avant la simulation.

Chapitre 7

Résultats

7.1 Remarques préliminaires

Le nombre d'occasions perdues en conflits de ressources sur l'unité entière avec un processeur superscalaire naturel a conduit à ajouter une nouvelle architecture munie de deux telles unités. L'analyse des résultats s'est révélée plutôt décevante : le gain en parallélisme est faible. Le nombre de conflits de ressources a fortement diminué mais la baisse a été compensée par une augmentation des conflits de données, notamment des RAW. Ceci montre que c'est la structure du programme qui limite le degré de parallélisme et non plus l'architecture. Les résultats des simulations concernant le processeur superscalaire avec une seule unité entière ne seront pas mentionnés dans la suite de cette étude et l'expression *processeur superscalaire* désignera le processeur muni de deux unités entières.

Quelques remarques sur la validité des résultats :

- Les simulations ont été effectuées sur un parc d'une dizaine de machines (SparcStation 1 à SparcStation 10) qui fonctionnent avec des versions de systèmes légèrement différentes. Ceci induit des variations sur les résultats, mais qui restent très faibles.
- Les gestionnaires d'exceptions ne sont pas tracés.
- Les appels à des bibliothèques dynamiques n'ont pas été optimisés.

7.2 Caractéristiques des programmes

Le tableau suivant résume les proportions des différents types d'instructions selon les programmes simulés et indique le taux de bonnes prédictions atteint par l'algorithme.

Programme		load	store	entier	flottant	saut	prédiction
Entier	cache	15 %	9 %	50 %		26 %	77 %
	xlisp	26 %	12 %	32 %		30 %	71 %
	eqntott	23 %	5 %	47 %		25 %	92 %
	compress	17 %	9 %	57 %		17 %	68 %
Moyenne entier		20 %	9 %	46 %		25 %	77 %
Flottant	mdljdp2	20 %	3 %	18 %	41 %	18 %	91 %
	wave5	17 %	6 %	41 %	17 %	19 %	80 %
	tomcatv	29 %	5 %	25 %	32 %	9 %	99,7 %
	alvinn	39 %	15 %	12 %	30 %	4 %	97 %
	swm256	17 %	7 %	38 %	21 %	17 %	92 %
Moyenne flottant		24,5 %	7 %	27 %	28 %	13,5 %	91,9 %

7.3 Performances

La métrique la plus simple utilisée pour évaluer les performances est l'IPC : le nombre moyen d'instructions séquencées à chaque cycle.

$$IPC = \frac{Nombre\ d'instructions\ exécutées}{Nombre\ de\ cycles\ nécessaires}$$

La figure 7.1 montre les performances atteintes sur un processeur superscalaire par les programmes qui ne font que des opérations entières en fonction du nombre d'étages de chargement. La figure 7.2 donne les valeurs pour les programmes de calcul flottant. Les résultats des simulations avec un processeur scalaire sont donnés sur les figures 7.3 et 7.4.

Trois constatations s'imposent immédiatement :

- Les performances sont nettement meilleures lorsque les branchements sont prédits. On vérifie que la prédiction de branchements a effectivement un intérêt. Le paragraphe 7.5 étudie plus précisément son impact sur le gain de performances.
- Le pipeline Mixte donne les meilleures performances. Ceci prouve qu'il est effectivement préférable de repousser le plus tard possible les étages d'exécution des instructions flottantes (afin de minimiser le temps d'attente après un chargement) alors les instructions entières doivent être terminées au plus tôt (pour réduire les délais de branchement et diminuer le temps perdu pour les calculs d'adresse).
  La différence entre les pipelines GE et Mixte étant uniquement la position des étages qui traitent les calculs flottants, il est normal que les programmes de calcul entier obtiennent les mêmes performances avec ces deux structures.
- Le degré de parallélisme diminue quand le nombre d'étages de chargement augmente. Plus le temps d'accès à une donnée est long, plus le nombre d'occasions perdues est grand, et par conséquent plus l'IPC est faible. On mesure là l'importance de la rapidité des mémoires.

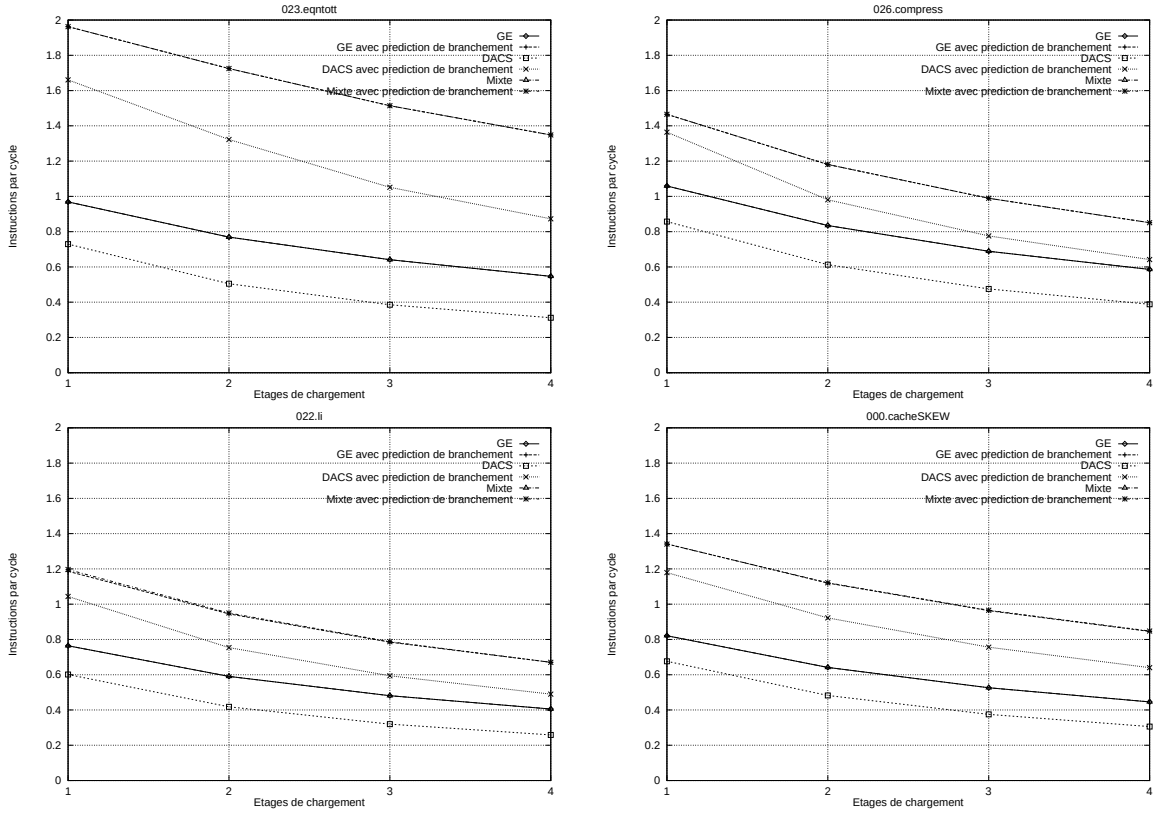

FIG. 7.1 – IPC pour calcul entier sur un processeur superscalaire de degré 5

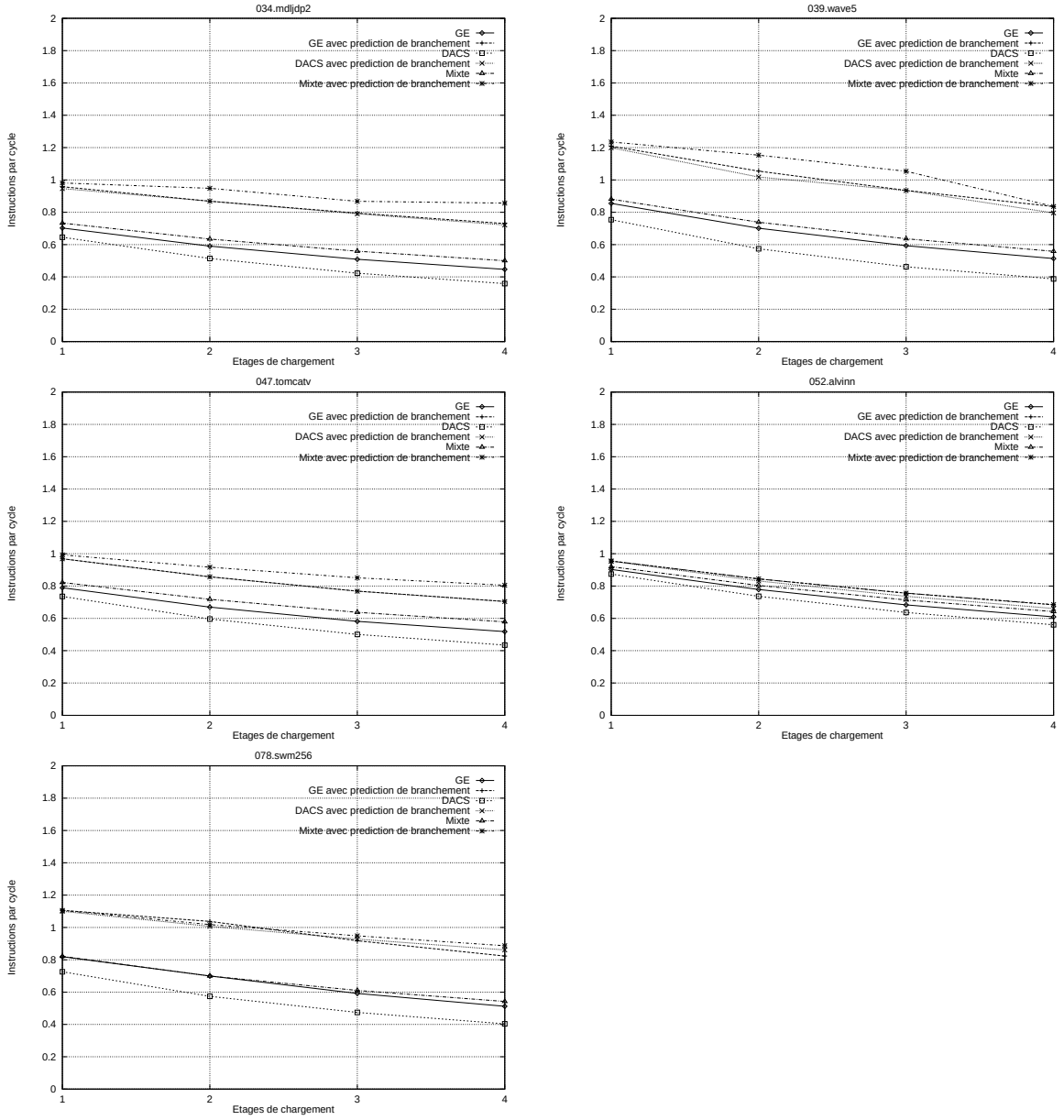

FIG. 7.2 – IPC pour calcul flottant sur un processeur superscalaire de degré 5

7.4 Gain apporté par le pipeline Mixte

Le pipeline Mixte permet d'atteindre un IPC nettement supérieur à celui obtenu par le pipeline DACS et légèrement supérieur à celui de GE dans le cas du calcul flottant (Mixte et GE sont identiques sur le calcul entier).

Le choix de la structure de pipeline Mixte est d'autant plus intéressante que le nombre d'étages de chargement est élevé, comme le montre la figure 7.5. Un gain de près de 60 % peut être apporté par rapport à un pipeline DACS sans prédiction de branchement.

- Le gain est moindre lorsque la prédiction de branchement est utilisée. En effet le nombre d'occasions perdues à cause des branchements est plus élevé avec un pipeline DACS qui détermine l'adresse où le contrôle doit être transféré très tard (cf. paragraphe 7.6.1). La prédiction diminue le nombre d'occasions perdues, ce qui explique que le gain soit plus faible.
- Les calculs flottants sont moins sensibles que les calculs entiers. La différence entre ces deux structures est l'étage d'exécution des instructions entières, qu'il est préférable de placer au plus tôt. Les programmes qui traitent des problèmes flottants contiennent en proportion

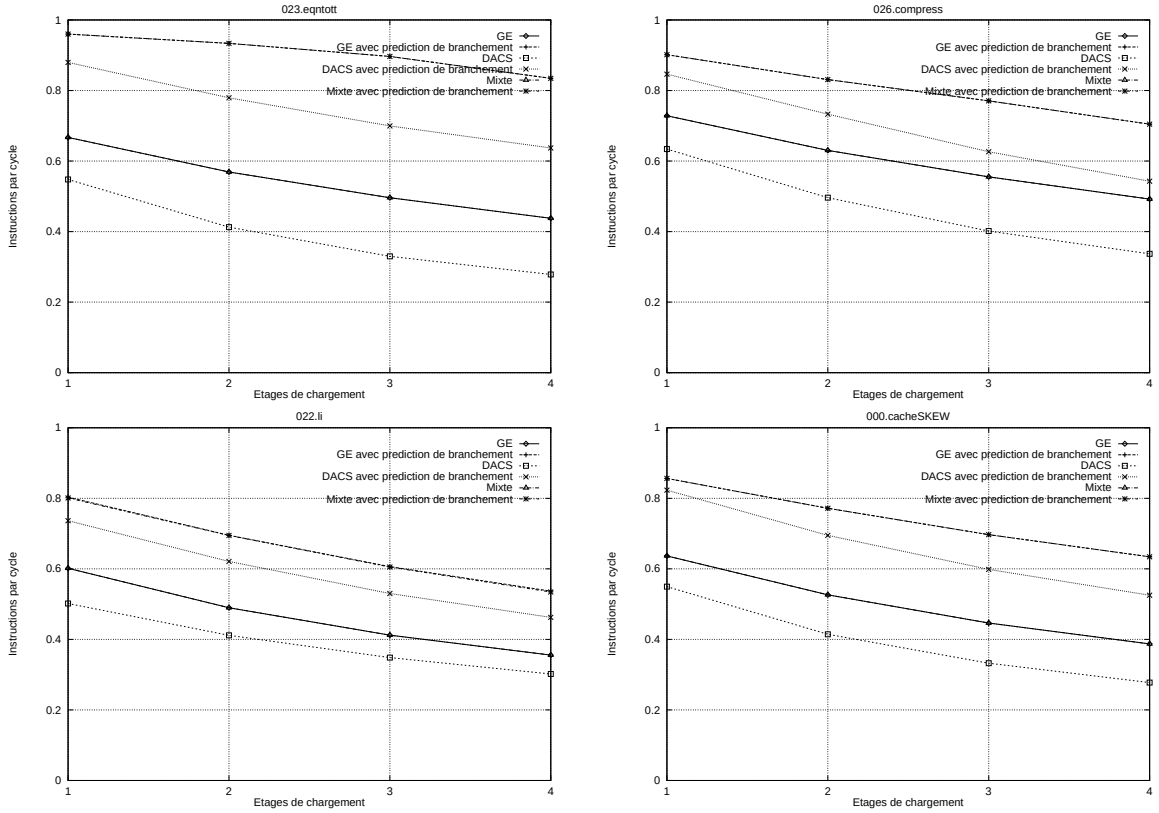

FIG. 7.3 – IPC pour calcul entier sur un processeur scalaire

beaucoup moins d'instructions entières et sont donc moins affectés par le déplacement de l'étage qui leur est dédié.

7.5 Impact de la prédiction de branchement

L'algorithme de prédiction de branchement mis en œuvre joue un grand rôle dans l'amélioration des performances. La figure 7.6 montre le gain de parallélisme (en pourcentage) apporté. Les barres représentent la moyenne des accélérations dans le cas du calcul entier et flottant, comparée à la moyenne sur tous les programmes. Le pipeline Mixte a le même comportement que le GE.

Il est à noter que les programmes qui ne font aucun calcul flottant réagissent beaucoup plus favorablement. Plusieurs raisons à cela : d'une part la proportion de branchements est plus élevée dans ce type de programmes d'où un grand nombre d'occasions perdues à cause des ruptures du flot de contrôle ; d'autre part les pertes liées aux dépendances de données sur les instructions flottantes sont plus lourdes que celles liées aux instructions entières (deux cycles perdus à chaque fois).

On remarque que le pipeline DACS est nettement aidé par la prédiction, ce qui s'explique par le choix de l'instant où l'adresse cible est connue (l'avant-dernier étage du pipeline). Il est aussi plus important d'avoir une prédiction sur un processeur superscalaire. L'impact d'une mauvaise prédiction est en effet plus important puisque le nombre d'occasions de séquencer perdues est plus élevé. Si l'on note N le nombre d'instructions du programme, d le degré du processeur ($d = 1$ pour un processeur scalaire) et α la proportion d'instructions qui entraîne une mauvaise prédiction, le nombre de cycles nécessaires à l'exécution du programme est approximativement $c_d(N) = \frac{N}{d} + \alpha \times N \times k$ si l'on l'on considère que k cycles sont perdus à chaque mauvaise prédiction. Le degré de parallélisme est alors :

$$IPC = \frac{N}{c_d(N)} = \frac{N}{\frac{N}{d} + \alpha N k} = \frac{d}{1 + \alpha d k}$$

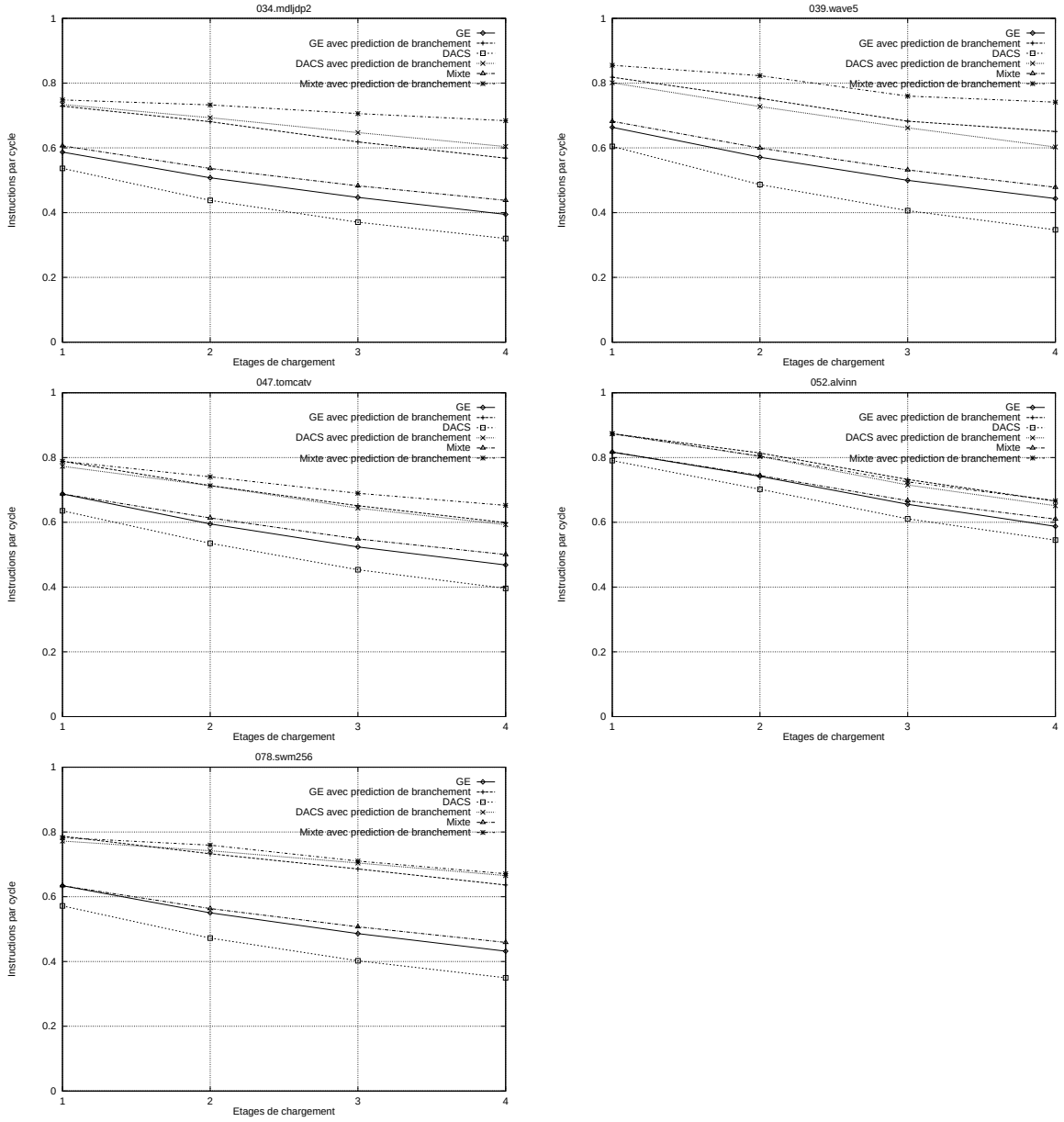

FIG. 7.4 – IPC pour calcul flottant sur un processeur scalaire

Le gain dû à la prédiction est :

$$G = \frac{IPC_{avec}}{IPC_{sans}} = \frac{1 + \alpha_{sans} dk}{1 + \alpha_{avec} dk} = \frac{1 + \beta dk}{1 + \alpha_{avec} dk}$$

qui croît lorsque le taux de mauvaises prédictions décroît (β représente la proportion de branchement).

7.6 Occasions perdues

Le détail de toutes les causes de pertes d'occasions est fournie en annexe B.

7.6.1 Conséquences des branchements

Un grand nombre d'occasions de séquencer sont perdues à la suite d'une mauvaise prédiction de branchement. Les pipelines GE et Mixte ont le même comportement par rapport aux branchements. Par contre le pipeline DACS est fortement pénalisé puisque l'adresse cible n'est

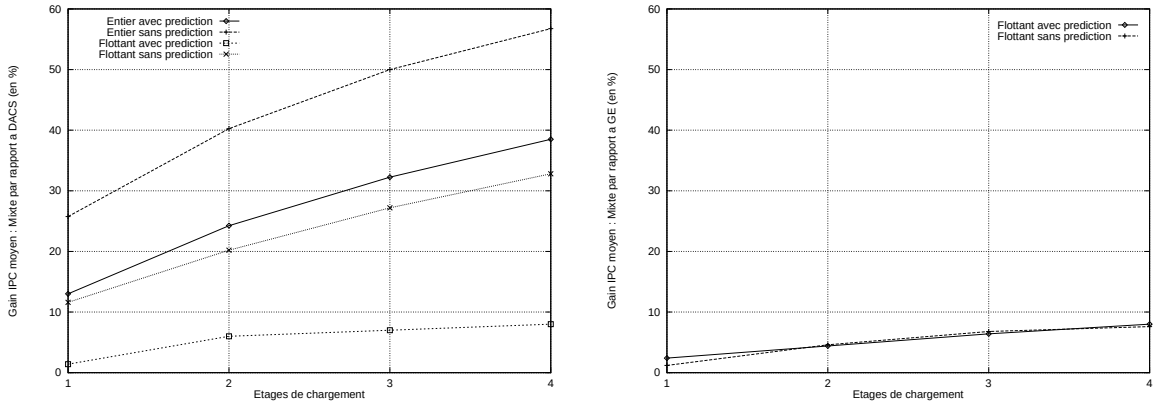

FIG. 7.5 – Gain apporté par le pipeline Mixte superscalaire

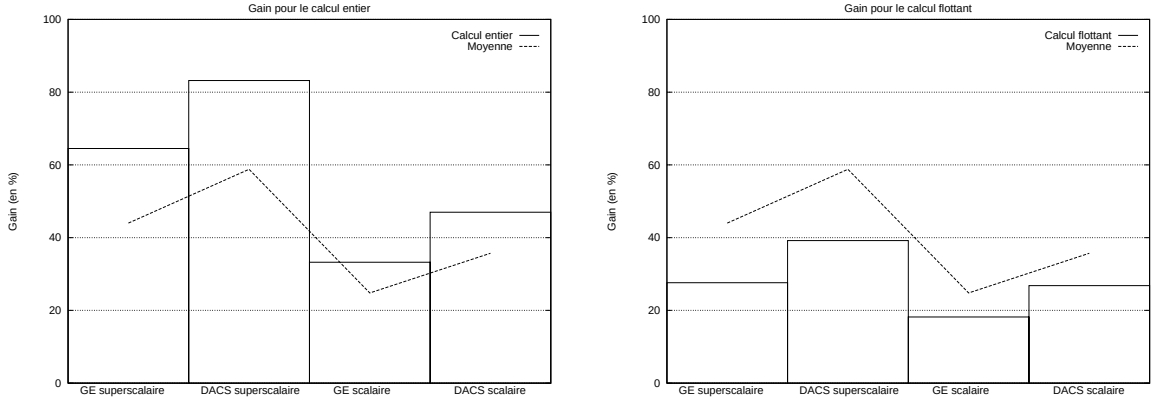

FIG. 7.6 – Gain apporté par la prédiction de branchement

connue que très tard. Les pertes sont similaires pour tous les programmes à l'exception de `alvinn` qui affiche des pertes beaucoup plus faibles. La figure 7.7 montre le nombre d'occasions perdues dans un cas typique (`compress`) pour un processeur superscalaire de degré cinq et un processeur scalaire.

Les pertes sont d'autant plus élevées que le nombre d'étages de chargement est grand et la croissance est linéaire. La pente est double pour le pipeline DACS. Ceci s'explique par le fait que l'augmentation du temps d'accès à la mémoire se traduit par l'ajout de deux étages dans le pipeline : un pour les instructions (IF) et un pour les données (MEM). Le pipeline DACS, qui détermine l'adresse cible au même cycle que la fin de l'accès à la mémoire, est à chaque fois pénalisé d'un temps double de celui des pipelines GE et Mixte.

Le cas de `alvinn` est différent : seulement 4 % des instructions sont des branchements et le taux de bonnes prédictions est supérieur à 97 %. Les délais de branchement sont donc réduits.

7.6.2 Aléas de structure

Les conflits de ressources sont une particularité des processeurs superscalaires. En effet un processeur scalaire ne peut séquencer qu'une instruction par cycle, toutes les unités fonctionnelles sont nécessairement libres. L'une d'entre elles va être utilisée mais dès le cycle suivant toutes sont à nouveau disponibles (on rappelle que l'on a supposé l'unité flottante pipelinée).

7.6.3 Dépendances de données

Les délais d'écriture correspondent au blocage d'une instruction `store` en attente de la donnée à écrire. De par la structure des pipelines GE et Mixte, ce type de délai ne peut se

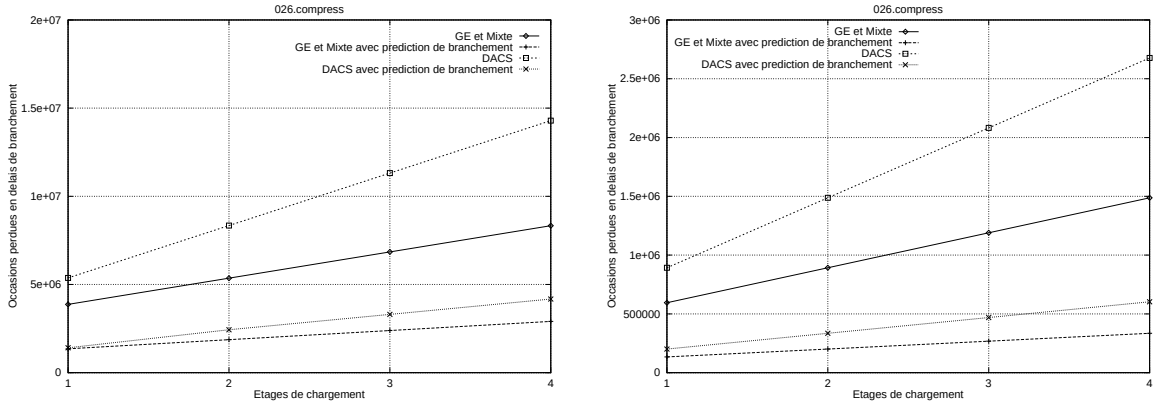

FIG. 7.7 – Évolution des occasions perdues à cause des branchements

produire avec des programmes qui ne font que du calcul entier : même si les deux instructions à l'origine de la dépendance sont séquencées dans le même groupe, aucun blocage ne se produit. Le `store` peut faire son calcul d'adresse à l'étage AC pendant que l'instruction entière calcule la valeur et celle-ci est immédiatement disponible au cycle suivant pour l'étage MEM. Seules les instructions flottantes peuvent donc produire de tels aléas dans une architecture basée sur les pipelines GE et Mixte. Le seul cas particulier de processeur DACS qui possède la même propriété est le processeur scalaire DACS avec un seul étage de chargement (cf. simulations 1 et 5, annexe B.5).

Les délais d'adressage n'existent pas avec un pipeline scalaire GE ou Mixte puisqu'une instruction entière fournit son résultat au cycle immédiatement suivant celui où elle a été séquencée, c'est-à-dire assez tôt même pour l'instruction immédiatement suivante (grâce au mécanisme de court-circuit).

Les délais de chargement sont inexistants dans l'architecture scalaire DACS. Leur nombre est considérablement réduit dans une architecture superscalaire DACS car ils proviennent uniquement du cas où deux instructions dépendantes sont candidates pour être séquencées dans le même groupe (cf. tableau 6.2 page 41).

Conclusion

Les simulations effectuées au cours de ce stage montrent l'importance de la structure du pipeline sur les performances d'un processeur. Il apparaît clairement que le pipeline Mixte est celui qui obtient les meilleurs résultats : il donne l'IPC le plus élevé dans tous les cas envisagés. Il est aussi à noter que la prédiction de branchement ne doit pas être négligée, surtout dans le cas de calcul entier où le gain de parallélisme est supérieur à 50 %. Le temps d'accès à une donnée en mémoire influe aussi fortement l'IPC : la perte varie entre 0,1 et 0,2 IPC par étage de chargement supplémentaire.

Toutes ces simulations ont utilisé un code optimisé en fonction de l'architecture cible et les instructions ont ensuite été exécutées dans l'ordre. Or un optimiseur de code, si efficace qu'il soit, ne dispose pas toujours de toutes les informations dont il pourrait avoir besoin pour aboutir au meilleur réordonnancement. Certaines données ne seront en effet connues qu'au moment de l'exécution. Il serait intéressant de voir dans quelle mesure le gain apporté par une exécution en désordre (*out-of-order*) peut justifier la complexité matérielle supplémentaire.

Quelques simulations supplémentaires avec un processeur utopique qui dispose de dix unités fonctionnelles de chaque type montrent que l'IPC maximum qu'il est possible d'atteindre est inférieur à 2,5. Dans ces simulations le nombre de conflits de ressources est nul et la performance est limitée uniquement par les dépendances de données et les délais de branchement. Un renommage dynamique des registres pourrait alors être envisagé pour améliorer l'utilisation des registres. Mais un inconvénient majeur de l'optimisation faite par *Oco* est qu'elle utilise un source en langage d'assemblage. Une grande partie des informations qui se trouvaient dans le langage de haut niveau est donc perdue et une véritable optimisation n'est pas possible. Une optimisation faite par le compilateur devrait aussi apporter des améliorations que le simple réordonnancement n'est pas capable de trouver.

Bibliographie

[ASU86] A. V. Aho, R. Sethi et J. D. Ullman : *Compilers Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[Bod89] François Bodin : *Optimisation de microcode pour une architecture horizontale et synchrone : Étude et mise en œuvre d'un compilateur.* Thèse de doctorat, Université de Rennes I, juin 1989.

[BYP+91] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales et Michael Shebanov : Single instruction stream parallelism is greater than two. *Dans ISCA*, pages 276–286, septembre 1991.

[Fis81] Joseph A. Fisher : Trace scheduling : A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–484, juillet 1981.

[HG83] John Hennessy et Thomas Gross : Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, juillet 1983.

[HP90] John L. Hennessy et David A. Patterson : *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 2e édition, 1990.

[Irl91] Gordon Irlam : Spa package. 1991.

[JS86] Yvon Jégou et André Seznec : Data synchronized pipeline architecture : Pipelining in multiprocessor environments. *Journal of Parallel and Distributed Computing*, 3: 508–526, 1986.

[JW89] Norman P. Jouppi et David W. Wall : Available instruction-level parallelism for superscalar and superpipelined machines. *Dans ASPLOS 3*, pages 272–282, avril 1989.

[Kri90] Sanjay M. Krishnamurthy : A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, juillet 1990.

[LGN92] Philip Lenir, R. Govindarajan et S. S. Nemawarkar : Exploiting instruction-level parallelism : The multithread approach. *Sigmicro Newsletter*, 23:189–192, décembre 1992.

[Log72] Luigi Logrippo : Renaming in program schemas. *Dans Proceeding of the IEEE 13th Annual Symposium on switching and Automata Theory*, pages 67–70, octobre 1972.

[LS92] P. Laporte et A. Seznec : *Une étude comparative des microprocesseurs MIPS R3000, Sun SPARC Version 7 et IBM Power : architectures et performances.* IRISA, février 1992.

[LW92] Monica S. Lam et Robert P. Wilson : Limits of control flow on parallelism. *Dans ISCA*, pages 46–57, juillet 1992.

[RF93] B. Ramakrishna Rau et Joseph A. Fisher : Instruction-level parallel processing : History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[SJH89] Michael D. Smith, Mike Johnson et Mark Horowitz : Limits on multiple instruction issue. *Dans ASPLOS 3*, pages 290–302, avril 1989.

[SKV93] André Seznec, Anne-Marie Kermarrec et Thierry Vauléon : Étude comparée des architectures des microprocesseurs MIPS R4000, DEC 21064 et TI SUPERSPARC. Rapport technique 1836, INRIA, janvier 1993.

[TF70] Garold S. Tjaden et Michael J. Flynn : Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, octobre 1970.

[TGH92] Kevin B. Theobald, Guang R. Gao et Laurie J. Hendren : On the limits of program parallelism and its smoothability. *Sigmicro Newsletter*, 23:10–19, décembre 1992.

[Uht91] Augustus K. Uht : A theory of reduced and minimal procedural dependencies. *IEEE Transactions on Computers*, 40(6):681–692, juin 1991.

[Uht93] Augustus K. Uht : Extraction of massive instruction level parallelism. *Computer Architecture News*, 21:5–12, juin 1993.

[Wal91] David W. Wall : Limits of instruction-level parallelism. *Dans ASPLOS 4*, volume 26, pages 176–188, avril 1991.

[WE93] Jian Wang et Christine Eisenbeis : Decomposed software pipelining : A new appoach to exploit instruction level parallelism for loop programs. *IFIP Transactions A [Computer Science and Technology]*, A-23:3–14, 1993.

Annexe A

IPC par programme et par configuration

Les tableaux suivant donnent l'IPC résultant des simulations dans tous les cas envisagés au cours de cette étude.

A.1 Processeur scalaire

A.1.1 Pipeline DACS

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,7341	0,5366	0,6931	0,438	0,6471	0,3705	0,6041	0,3202
wave5	0,801	0,6046	0,7275	0,4865	0,6618	0,4062	0,6028	0,347
tomcatv	0,7729	0,6356	0,713	0,5353	0,6438	0,4535	0,5926	0,3959
alvinn	0,8733	0,7906	0,8033	0,702	0,7153	0,6107	0,6512	0,5451
swm256	0,772	0,5716	0,7421	0,4724	0,7041	0,4022	0,6653	0,3498
eqntott	0,8797	0,5479	0,7794	0,4127	0,6997	0,3298	0,6372	0,2785
compress	0,8464	0,6341	0,7329	0,4961	0,6265	0,4016	0,5429	0,3369
xlisp	0,7366	0,5018	0,621	0,4116	0,5301	0,3483	0,4624	0,3019
cache	0,8231	0,5496	0,695	0,4144	0,5985	0,3325	0,5253	0,2775

A.1.2 Pipeline GE

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,729	0,5868	0,6809	0,5075	0,6184	0,447	0,5684	0,3945
wave5	0,8182	0,6635	0,753	0,5712	0,6825	0,4997	0,6506	0,4435
tomcatv	0,7886	0,6876	0,7133	0,5948	0,6511	0,524	0,5989	0,4683
alvinn	0,8737	0,8168	0,814	0,7418	0,7323	0,6557	0,6654	0,5875
swm256	0,7868	0,6339	0,7327	0,5503	0,6857	0,4862	0,6363	0,4319
eqntott	0,96	0,6671	0,9336	0,5691	0,8965	0,4959	0,8346	0,4378
compress	0,9016	0,7284	0,831	0,6301	0,7706	0,5552	0,7046	0,4922
xlisp	0,8032	0,6016	0,6952	0,4897	0,6062	0,412	0,5369	0,3555
cache	0,8563	0,6366	0,7718	0,5263	0,6969	0,4463	0,6343	0,3873

A.1.3 Pipeline Mixte

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,7482	0,6064	0,7327	0,5365	0,7058	0,4827	0,6841	0,438
wave5	0,8551	0,6819	0,823	0,5994	0,7599	0,532	0,7412	0,4784
tomcatv	0,7886	0,6876	0,7404	0,6135	0,6897	0,5487	0,6524	0,5004
alvinn	0,8739	0,8169	0,8043	0,7446	0,7248	0,6669	0,6665	0,6098
swm256	0,7823	0,6341	0,7593	0,5635	0,7101	0,5071	0,6711	0,4589
eqntott	0,96	0,6671	0,9336	0,5691	0,8965	0,4959	0,8346	0,4378
compress	0,9016	0,7284	0,831	0,6301	0,7706	0,5552	0,7046	0,4922
xlisp	0,801	0,6016	0,6948	0,4897	0,6052	0,412	0,5345	0,3555
cache	0,8563	0,6366	0,7718	0,5263	0,6969	0,4463	0,6343	0,3873

A.2 Processeur superscalaire de degré cinq

A.2.1 Pipeline DACS

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,9462	0,6456	0,8689	0,5142	0,7906	0,4228	0,7214	0,3585
wave5	1,1993	0,7541	1,0178	0,5737	0,9345	0,4629	0,7959	0,3875
tomcatv	0,9682	0,7355	0,8566	0,5957	0,7679	0,5005	0,7041	0,4347
alvinn	0,9537	0,8741	0,8311	0,7365	0,7363	0,6364	0,6609	0,5602
swm256	1,1002	0,7259	1,007	0,574	0,9283	0,4741	0,861	0,4038
eqntott	1,6604	0,7293	1,3225	0,5048	1,0515	0,3855	0,8727	0,3118
compress	1,3639	0,8571	0,9807	0,6127	0,7758	0,475	0,6417	0,3879
xlisp	1,0436	0,6013	0,7537	0,4181	0,5938	0,3196	0,4898	0,2587
cache	1,1789	0,6762	0,9223	0,4825	0,756	0,375	0,64	0,3061

A.2.2 Pipeline GE

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,9582	0,7027	0,8683	0,5905	0,7944	0,5092	0,7301	0,4468
wave5	1,2089	0,855	1,0544	0,7013	0,9352	0,593	0,835	0,5136
tomcatv	0,9686	0,7892	0,8571	0,6695	0,7686	0,5813	0,7048	0,5181
alvinn	0,9549	0,9028	0,844	0,7783	0,7555	0,6835	0,6838	0,6093
swm256	1,1057	0,8192	1,0373	0,7	0,9179	0,5918	0,8232	0,5125
eqntott	1,9623	0,9688	1,7248	0,7693	1,5139	0,6409	1,3485	0,5468
compress	1,4655	1,0591	1,1811	0,8346	0,9892	0,6886	0,8509	0,5861
xlisp	1,1867	0,7642	0,9452	0,5903	0,7849	0,4809	0,6704	0,4057
cache	1,3418	0,8209	1,1195	0,6412	0,9635	0,526	0,8456	0,4458

A.2.3 Pipeline Mixte

Étages	1		2		3		4	
Prédiction	Avec	Sans	Avec	Sans	Avec	Sans	Avec	Sans
mdljdp2	0,9813	0,7329	0,948	0,6344	0,8676	0,5593	0,8566	0,5
wave5	1,2348	0,8814	1,1531	0,7383	1,0537	0,6358	0,8357	0,5584
tomcatv	0,9928	0,8219	0,9165	0,7179	0,851	0,6373	0,8049	0,5785
alvinn	0,9549	0,9196	0,8441	0,8036	0,7563	0,7136	0,685	0,6416
swm256	1,1054	0,8212	1,018	0,6992	0,9481	0,6107	0,8872	0,5421
eqntott	1,9623	0,9688	1,7248	0,7693	1,5139	0,6409	1,3485	0,5468
compress	1,4654	1,0591	1,1811	0,8346	0,9892	0,6886	0,8509	0,5861
xlisp	1,1963	0,7642	0,9498	0,5903	0,7874	0,4809	0,6703	0,4057
cache	1,3408	0,8212	1,1217	0,6415	0,9657	0,5262	0,8477	0,4461

Annexe B

Détail des occasions perdues

Les tableaux de cette annexe recensent, pour chaque simulation, l'origine des pertes d'occasions.

Les simulations 1 à 4 correspondent aux pipelines possédant de 1 à 4 étages de chargement avec une prédiction de branchement ; les simulations 5 à 8 n'utilisent pas la prédiction.

Numéro	Nombre d'étages	Prédiction
1	1	oui
2	2	oui
3	3	oui
4	4	oui
5	1	non
6	2	non
7	3	non
8	4	non

B.1 Pipeline GE superscalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1070912	1135004	352531	1051416	1219226	861144	891304	980562	617063
adressage	218736	198544	354512	429294	232041	25004	95659	66	68892
écriture	0	0	0	0	303506	227345	858069	2157151	328371
branchement	1604447	1847410	394755	1354500	374644	803306	6469	28665	319689
conflits	1359524	1522615	1217077	943458	2347565	1638405	3059278	3100647	1700258

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1905152	2188443	759685	1853152	1999406	1476174	2005334	1737253	1142622
adressage	219246	198279	354187	429294	232833	59176	95659	66	68892
écriture	0	0	0	0	303506	227782	857493	2024927	328167
branchement	2253150	2547050	544319	1872625	532985	1203628	9004	39785	442527
conflits	1355659	1522857	1229533	1059628	2347829	1650873	3059650	3449657	1758990

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	2742474	3248226	1259228	2777524	2996013	2168107	2992418	2677060	2065152
adressage	219246	194131	355236	429294	232833	54801	95656	66	68892
écriture	0	0	0	0	303506	228509	857799	2190198	328371
branchement	2861985	3249365	694556	2390750	681032	1527501	11539	50913	566159
conflits	1355647	1522587	1228666	1053176	2348365	1696469	3313878	3649029	1804233

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	3580094	4318696	1764323	3695444	4106983	2930585	4106839	3736576	3010252
adressage	219246	191199	355236	429294	232833	41469	95659	66	68892
écriture	0	0	0	0	303506	228509	858312	2190213	328371
branchement	3470820	3951680	845511	2908875	829077	1853638	14074	62592	689394
conflits	1355647	1521190	1228661	1053176	2349421	1733084	3313627	3884052	1826888

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1080243	797200	135369	870894	1219238	883551	891305	982737	524429
adressage	224618	292541	403945	477793	239035	28762	191297	66	68892
écriture	0	0	0	0	303506	182045	858069	1855207	236659
branchement	6834714	6380295	5082472	3873739	4085593	4255351	2250360	996968	3796744
conflits	760339	909854	936892	761967	1930765	1291280	2948001	3007372	1473952

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1890411	1497850	234945	1488136	1999428	1428461	2005335	1740493	879583
adressage	224578	292541	443341	477793	239827	62933	191297	66	68892
écriture	0	0	0	0	303506	182045	857493	1722983	236506
branchement	9440775	8814360	7113526	5361140	5654216	5875620	3128385	1380888	5274634
conflits	757933	909854	936896	878144	1933405	1319959	2948373	3356382	1532787

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	2701157	2198500	402297	2228008	2996045	2045044	2992329	2679270	1631708
adressage	224578	292541	403945	477793	239827	58492	191297	66	68892
écriture	0	0	0	0	303506	182045	857745	1888254	236659
branchement	12046840	11248425	9092052	6848540	7222047	7495889	4006410	1764816	6752524
conflits	757933	909854	934848	871692	1934725	1365842	3202488	3555754	1577927

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	3511887	2899150	571697	2961428	4107025	2671690	4106840	3737728	2406403
adressage	224578	292541	403945	477793	239827	45160	191297	66	68892
écriture	0	0	0	0	303506	182045	858312	1888254	236659
branchement	14652905	13682490	11096842	8335940	8790142	9116160	4884435	2148732	8230414
conflits	757933	909854	934848	871692	1935781	1402463	3202350	3790767	1600582

B.2 Pipeline DACS superscalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	360644	258999	55044	484692	533868	271280	509245	942973	213987
adressage	980987	1290101	1107725	995780	580982	252936	526578	1758	260841
écriture	221429	202321	96502	305250	489326	343644	1335320	3145479	606258
branchement	2330101	2343321	609393	1410362	541677	1109970	9004	39787	442905
conflits	1382529	1559882	1206117	968888	2448643	1617323	2375765	2233140	1662764

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	359084	259828	58828	513086	558644	274140	509242	1044151	232043
adressage	1764369	2430953	1845243	1784228	1146903	541486	1162994	124402	435151
écriture	324053	298007	219391	615262	466398	521709	1811634	4052445	895526
branchement	3591932	3845248	831741	2436127	843037	2055751	14074	61290	688815
conflits	1393746	1577618	1315056	1226634	2414086	1620693	2503421	2294511	1667277

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	362597	260398	124364	519538	558644	293616	509242	1044390	232043
adressage	2551172	3592417	2858175	2621618	1722437	828798	2086487	277053	609461
écriture	344096	359888	344285	996852	492758	684665	2288529	5256755	1207704
branchement	4852812	5210746	1129959	3306327	1095065	2254866	19144	83278	935085
conflits	1389864	1577606	1286146	1248579	2418312	1579568	2566949	2327511	1662770

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	364724	261015	124364	519538	558644	312273	509238	1044386	232039
adressage	3349163	4747217	3902875	3484816	2297187	1122465	2882503	429753	806320
écriture	350781	422928	472245	1284872	525048	838340	2511256	5828168	1519899
branchement	6113692	6576256	1426129	4176527	1383235	3484113	24214	105254	1181355
conflits	1389864	1577704	1286141	1255031	2419896	1578659	2535671	2327513	1662763

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	372303	206885	50115	418871	533872	370139	604504	912855	177279
adressage	652598	1114788	1143353	962816	454510	248030	622395	120266	155007
écriture	212401	205795	97894	305250	489326	261109	1335320	2654838	368143
branchement	9466793	8834011	7091206	5373016	5647758	5876153	3128372	1380915	5274634
conflits	798377	926877	921665	815658	1961151	1257357	2169019	2201022	1480379

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	372303	206885	50115	447269	558648	372969	604501	1014039	195335
adressage	1098565	1905986	1848104	1633970	808761	528964	1258811	270484	263692
écriture	278619	271879	220771	466083	466398	403822	1811634	3373423	476777
branchement	14678921	13701810	11123344	8341364	8784212	9117471	4884422	2148743	8230414
conflits	798986	954079	989346	1027344	1927650	1264977	2296675	2232192	1485012

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	372309	206885	50115	453721	558648	393245	604501	1013987	213391
adressage	1542545	2742761	2641124	2376100	1169206	812110	2182304	420700	372377
écriture	291715	360469	343621	792623	491438	525981	2288529	4389488	608185
branchement	19891051	18569940	15132924	11316164	11920402	12358012	6640472	2916591	11186194
conflits	798980	954079	989351	1049289	1929762	1219033	2360203	2265192	1480515

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	379171	206885	50115	453721	558648	411032	604497	1013984	213391
adressage	2047989	3579536	3434149	3144038	1530971	1096831	2978320	570970	481062
écriture	307621	449059	466471	1053328	523728	637109	2511256	4885468	739610
branchement	25104797	23438070	19142504	14290964	15056592	15598552	8396522	3684427	14141974
conflits	808344	954079	989351	1055741	1931346	1219363	2328925	2265196	1480515

B.3 Pipeline Mixte superscalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1065078	1135129	352531	1051288	578977	404497	317761	942453	341863
adressage	218264	192274	354512	429281	232014	48616	95664	66	86531
écriture	0	0	0	0	489326	367244	1334354	3145413	606026
branchement	1615347	1794044	394755	1354570	341849	931325	6469	28665	319672
conflits	1361722	1531862	1217077	943460	2344037	1608843	2565859	2234069	1638370

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1894949	2191753	759685	1852909	811818	505269	476828	1090401	532217
adressage	221051	191776	354187	429281	232569	51768	95664	66	86531
écriture	0	0	0	0	464550	511541	1810323	4052607	918287
branchement	2241168	2509268	544319	1872720	362549	1104015	9004	39785	443202
conflits	1358035	1526265	1229533	1059630	2359042	1651588	2693419	2265159	1660890

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	2726883	3285422	1259228	2777166	1053229	736126	921639	1137255	704447
adressage	223499	188896	355236	429281	227358	51990	95664	66	86531
écriture	0	0	0	0	489326	651213	2286873	5257282	1230562
branchement	2846973	3183482	694556	2390870	1036158	1469084	11539	50913	566517
conflits	1358933	1526834	1228666	1053178	2322240	1683856	2756901	2298412	1683543

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	3560353	4317499	1764323	3694971	1299322	1100594	1493920	1850379	876677
adressage	223499	190999	355236	429281	232556	51650	95664	66	86531
écriture	0	0	0	0	522060	772105	2509439	5828801	1542837
branchement	3452778	3954403	845511	2909020	931702	3319958	14074	62592	689832
conflits	1357717	1520525	1228661	1053178	2321867	1559629	2725271	2299153	1701660

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1077067	797193	135369	870766	578984	457004	317762	912862	305153
adressage	224648	292549	403945	477741	239035	50992	191310	66	68892
écriture	0	0	0	0	489326	283172	1334354	2654754	367919
branchement	6834714	6380296	5082472	3873757	4079399	4255714	2250370	996968	3796743
conflits	761341	909866	936892	761961	1905989	1270652	2406776	2171490	1455987

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	1882255	1497838	234945	1487933	811830	593224	476829	1060810	495439
adressage	226532	292549	443341	477741	239563	54048	191310	66	68892
écriture	0	0	0	0	464550	387373	1810323	3373233	499349
branchement	9440779	8814361	7113526	5361158	5648022	5875984	3128395	1380888	5274633
conflits	760369	909866	936896	878138	1908893	1302672	2534336	2202580	1478642

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	2687453	2198483	402297	2227730	1053246	758337	921640	1107664	667669
adressage	228980	292549	403945	477741	239563	54052	191310	66	68892
écriture	0	0	0	0	489326	490411	2286873	4389193	630779
branchement	12046844	11248426	9092052	6848558	7215853	7496254	4006420	1764816	6752523
conflits	761273	909866	934848	871686	1916143	1327459	2597818	2235833	1501297

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	3493883	2899128	571697	2961075	1297302	946057	1493921	1707659	839899
adressage	228980	292549	403945	477741	239827	54055	191310	66	68892
écriture	0	0	0	0	520296	571034	2509439	4885068	762209
branchement	14652909	13682491	11096842	8335958	8783948	9116141	4884445	2148732	8230413
conflits	760057	909866	934848	871686	1915879	1326208	2566188	2236549	1519421

B.4 Pipeline GE scalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	99071	103636	5894	56813	110766	70937	64	1089	52922
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	8888	92978	76474	21744
branchement	236408	294784	61434	133926	39214	141432	1012	3054	50756
conflits	0	0	0	0	0	0	0	0	0

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	236422	281842	25778	154734	303209	175132	127275	119248	181612
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	8888	92978	107991	21744
branchement	354843	431340	89238	200874	58065	220332	1518	4581	76134
conflits	0	0	0	0	0	0	0	0	0

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	396697	487051	56334	252657	512222	295919	285481	270893	310338
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	9214	92978	175687	26258
branchement	473124	569832	130452	267832	169856	341068	2024	6108	101512
conflits	0	0	0	0	0	0	0	0	0

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	561607	691014	143625	398340	733625	418699	536851	457446	472684
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	10834	92978	175687	26275
branchement	591405	712290	177185	334790	212320	341425	2530	7635	126890
conflits	0	0	0	0	0	0	0	0	0

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	99071	103636	5894	56813	110766	70937	64	1089	52922
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	8888	92978	76474	21744
branchement	1042426	973626	801916	594960	627238	648108	351210	153568	591156
conflits	0	0	0	0	0	0	0	0	0

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	236155	234943	23203	133787	303209	173359	127275	119039	165215
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	8888	92978	107991	21744
branchement	1563639	1460439	1202874	892440	940857	972162	526815	230352	886734
conflits	0	0	0	0	0	0	0	0	0

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	395847	374694	42128	210763	512225	291772	285481	270475	277508
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	9214	92978	175687	26258
branchement	2084852	1947252	1603832	1189920	1254476	1296216	702420	307136	1182312
conflits	0	0	0	0	0	0	0	0	0

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	556765	514820	74585	316145	733629	412037	536851	456579	423421
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	30970	10834	92978	175687	26275
branchement	2606065	2434065	2004790	1487400	1568095	1620270	878025	383920	1477890
conflits	0	0	0	0	0	0	0	0	0

B.5 Pipeline DACS scalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	75038	154472	133124	116393	43391	9752	48081	23	17224
écriture	0	0	0	0	61940	18025	154964	184557	48002
branchement	354612	427146	88287	200889	58821	252183	1518	4581	89490
conflits	0	0	0	0	0	0	0	0	0

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	234111	374856	276057	266701	102564	53292	127342	52	34448
écriture	52107	34870	24740	35655	105298	35433	247942	360509	79617
branchement	591420	572672	155830	334790	96775	412380	2530	7635	126890
conflits	0	0	0	0	0	0	0	0	0

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	394484	615772	436316	453412	211321	99812	254679	30117	56186
écriture	75692	64769	50486	112776	92910	59346	372038	574853	129360
branchement	827988	715840	204981	468706	135485	580580	3542	10689	177646
conflits	0	0	0	0	0	0	0	0	0

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	555101	849999	561248	653033	326269	159299	381828	60507	104590
écriture	78871	79044	76223	176659	99368	96033	434088	713687	191812
branchement	1064556	859008	263538	602622	174195	743598	4554	13743	228402
conflits	0	0	0	0	0	0	0	0	0

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	75038	154472	133124	116393	43391	9752	48081	23	17224
écriture	0	0	0	0	61940	17854	154964	184549	48002
branchement	1563639	1460439	1202874	892440	940857	972162	526815	230352	886734
conflits	0	0	0	0	0	0	0	0	0

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	166989	333003	273054	252954	102564	52935	127342	48	34448
écriture	52107	34870	24740	35655	105298	34945	247942	322762	74277
branchement	2606065	1947252	2004790	1487400	1568095	1620270	878025	383920	1477890
conflits	0	0	0	0	0	0	0	0	0

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	259544	512799	430303	425918	168459	99041	254679	30087	56186
écriture	63410	54188	50483	89863	92910	57572	372038	499367	100552
branchement	3648491	2434065	2806706	2082360	2195333	2268378	1229235	537488	2069046
conflits	0	0	0	0	0	0	0	0	0

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	0	0	0	0	0	0	0	0	0
adressage	352194	686073	499379	611792	240548	156992	381828	60137	86952
écriture	66585	75009	76220	137617	99368	90820	434088	600462	126827
branchement	4690917	2920878	3608622	2677320	2822571	2916486	1580445	691056	2660202
conflits	0	0	0	0	0	0	0	0	0

B.6 Pipeline Mixte scalaire

Simulation 1

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	99071	103638	5894	56813	264	1931	1	5	17224
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	61940	16780	154964	184557	48002
branchement	236408	300462	61434	133926	73578	119896	1012	3054	59660
conflits	0	0	0	0	0	0	0	0	0

Simulation 2

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	236422	282175	25778	154734	528	18637	30995	9098	34448
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	105298	26356	247942	360487	79617
branchement	354843	432219	89238	200874	63036	167325	1518	4581	73860
conflits	0	0	0	0	0	0	0	0	0

Simulation 3

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	396697	489930	56334	252657	48336	49136	61990	18196	51672
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	92910	43370	372038	574810	123720
branchement	473124	571264	130452	267832	115268	290168	2024	6108	166596
conflits	0	0	0	0	0	0	0	0	0

Simulation 4

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	561607	693880	143625	398340	96144	80993	124164	103212	86119
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	99104	64099	434088	713623	185774
branchement	591405	722645	177185	334790	129790	296565	2530	7635	208245
conflits	0	0	0	0	0	0	0	0	0

Simulation 5

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	99071	103638	5894	56813	264	1931	1	5	17224
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	61940	16602	154964	184549	43488
branchement	1042426	973626	801916	594960	627238	648108	351210	153568	591156
conflits	0	0	0	0	0	0	0	0	0

Simulation 6

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	236155	234946	23203	133787	528	18564	30995	9098	34448
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	105298	25948	247942	322740	69763
branchement	1563639	1460439	1202874	892440	940857	972162	526815	230352	886734
conflits	0	0	0	0	0	0	0	0	0

Simulation 7

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	395847	374698	42128	210763	48336	48925	61990	18196	51672
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	92910	37236	372038	499324	96038
branchement	2084852	1947252	1603832	1189920	1254476	1296216	702420	307136	1182312
conflits	0	0	0	0	0	0	0	0	0

Simulation 8

	cache	xlisp	eqntott	compress	mdljdp2	wave5	tomcatv	alvinn	swm256
chargement	556765	514825	74585	316145	96144	80613	124164	102972	86119
adressage	0	0	0	0	0	0	0	0	0
écriture	0	0	0	0	99104	54095	434088	600398	122313
branchement	2606065	2434065	2004790	1487400	1568095	1620270	878025	383920	1477890
conflits	0	0	0	0	0	0	0	0	0